

안전한 Clean Code 작성을 위한 C 시큐어 코딩 실무

강사 : 정혜경, 김기희
(교재 제작 : 렉토피아)

▶ 다음 코드의 결과를 예측하시요.

```
#include <stdio.h>

int main()
{
    int x = -1;
    unsigned int y = 1;

    if (x >= y)
    {
        printf("True\n");
    }
    else
    {
        printf("False\n");
    }

    getchar();
    return 0;
}
```

PRE-31. 매크로 함수 호출 시 주어지는 인자에 대한 증가, 감소, 메모리 변수 접근 등은 부수적인 효과(Side Effect)를 발생시킬 수 있다.

[문제코드]

```
#include <stdio.h>
#define CUBE(x) ((x) * (x) * (x))

int main()
{
    int i = 2;
    int a = CUBE(++i);

    printf("a = %d\n", a);
    getchar();
    return 0;
}
```

[해결방법] inline 함수를 이용하여 해결

```
#include <stdio.h>

inline int cube(int x)
{
    return x * x * x;
}

int main()
{
    int i = 2;
    int a = cube(++i);

    printf("a = %d\n", a);
    return 0;
}
```

[문제코드] 매크로가 사용하는 변수명과 매크로가 사용되는 블록내의 변수명이 같을 경우, 변수 사용에 문제가 발생할 수 있다.

```
#include <stdio.h>

size_t count = 0;
#define EXEC_BUMP(func) (func(), ++count)

void g(void)
{
    printf("g() 호출, count = %u.\n", ++count);
}

void aFunc(void)
{
    size_t count = 0;
    while (count++ < 10)
    {
        EXEC_BUMP(g);
    }
}

int main()
{
    aFunc();
    getchar();
    return 0;
}
```

[해결방법] inline 함수를 이용하여 해결한다.

```
#include <stdio.h>

size_t count = 0;

void g(void)
{
    printf("g() 호출, count = %u.\n", ++count);
}

typedef void (*exec_func)(void);
inline void exec_bump(exec_func f)
{
    f();
    ++count;
}

void aFunc(void)
{
    size_t count = 0;
    while (count++ < 10)
```

```
        {
            exec_bump(g);
        }
    }

int main()
{
    aFunc();
    getchar();
    return 0;
}
```

PRE-01. 매크로 함수의 치환부에서는 매개변수에 괄호를 사용하라.

[문제코드]

```
#include <stdio.h>
#define CUBE(I) (I * I * I)

int main()
{
    int a = CUBE(2 + 1);
    printf("a = %d\n", a);

    return 0;
}
```

[해결방법] 치환목록의 매개변수를 ()로 묶어준다.

```
#include <stdio.h>
#define CUBE(I) ((I) * (I) * (I))

int main()
{
    int a = CUBE(2 + 1);
    printf("a = %d\n", a);

    return 0;
}
```

[문제코드] getchar() FINAL 는 getchar() -1로 치환되어 부적절하게 평가된다.

```
#include <stdio.h>
#define FINAL -1

int main()
{
    char ch;

    while ((ch = getchar()) FINAL)
    {
        printf("%c", ch);
        printf("[%d]\n", ch);
    }

    getchar();
    return 0;
}
```

[해결방법] 치환될 영역은 반드시 괄호로 둘러싸거나 열거형(enum)상수로 치환한다.

```
#include <stdio.h>
enum { FINAL = -1 };
// 또는 #define FINAL (-1)

int main()
{
    char ch;

    while ((ch = getchar()) != FINAL)
    {
        printf("%c", ch);
    }

    getchar();
    return 0;
}
```

PRE-03. 타입 재정의의 시, 매크로 정의 대신 타입 정의를 사용하라.

자료형을 재정의 할 때에는 매크로 대신 타입정의(`typedef`)를 사용하라.

[문제코드] 포인터 타입을 매크로로 정의하면 부작용을 초래한다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h> // gcc 컴파일 시 필요

#define BUFFSIZE 100
#define cstring char *

cstring getString(const char *);

int main()
{
    cstring name, tel;
    name = getString("이름");
    tel = getString("전화");
    printf("입력된 이름 : %s\n", name);
    printf("입력된 전화번호 : %s\n", tel);
    free(name);
    free(tel);
    getchar();
    return 0;
}

cstring getString(const char *quest)
{
    cstring nr, buf = NULL;
    char tmp[BUFFSIZE];
    printf("%s : ", quest);
    if ((fgets(tmp, sizeof(tmp), stdin)) != NULL)
    {
        nr = strchr(tmp, '\n');
        if(nr!=NULL) // 개행문자 삭제
        {
            *nr = '\0';
        }
        else // 초과 입력된 데이터 삭제
        {
            __fpurge(stdin); // Visual Studio인 경우 fflush(stdin)
        }
        buf = (char *)calloc(strlen(tmp) + 1, sizeof(char));
        strncpy(buf, tmp, strlen(tmp));
    }

    return buf;
}

```

[해결방법] 타입정의 typedef로 해결한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h> // gcc 컴파일 시 필요

#define BUFFSIZE 100
typedef char * cstring;

cstring getString(const char *);

int main()
{
    cstring name, tel;

    name = getString("이름");
    tel = getString("전화");

    printf("입력된 이름 : %s\n", name);
    printf("입력된 전화번호 : %s\n", tel);

    free(name);
    free(tel);
    getchar();
    return 0;
}

cstring getString(const char *quest)
{
    cstring nr, buf = NULL;
    char tmp[BUFFSIZE];

    printf("%s : ", quest);
    if ((fgets(tmp, sizeof(tmp), stdin)) != NULL)
    {
        nr = strchr(tmp, '\n');
        if(nr!=NULL) // 개행문자 삭제
        {
            *nr = '\0';
        }
        else // 초과 입력된 데이터 삭제
        {
            __fpurge(stdin); // Visual Studio인 경우 fflush(stdin)
        }
        buf = (char *)calloc(strlen(tmp) + 1, sizeof(char));
        strncpy(buf, tmp, strlen(tmp));
    }
    return buf;
}
```

PRE-10. 복수 구문의 매크로는 do ~ while 루프로 감싼다.

[문제코드]

```
#include <stdio.h>
#define SWAP(x,y) \
    tmp = x; \
    x = y; \
    y = tmp

int main()
{
    int x, y, z, tmp = 0;

    printf("x = "); scanf("%d", &x);
    printf("y = "); scanf("%d", &y);
    printf("z = "); scanf("%d", &z);

    if (z == 0) SWAP(x, y);
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#define SWAP(x,y) \
do { \
    tmp = x; \
    x = y; \
    y = tmp; \
} while (0)

int main()
{
    int x, y, z, tmp = 0;

    printf("x = "); scanf("%d", &x);
    printf("y = "); scanf("%d", &y);
    printf("z = "); scanf("%d", &z);

    if (z == 0) SWAP(x, y);
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    return 0;
}
```

DCL-05. 코드의 가독성을 높이기 위해 타입 정의를 사용하라.

☞ signal()함수는 아래와 같이 선언되어 있다.

```
void (*signal(int signum, void (*handler)(int)))(int);
```

이것은 signal()함수는 아래와 같이 두개의 파라미터를 가지며

```
signal( int signum, void (*handler)(int) )
```

signal()함수가 반환하는 값은 아래와 같은 함수포인터형임을 나타낸다.

```
void (*)(int);
```

위의 signal()함수는 아래와 같이 typedef를 이용하여 함수 포인터 타입을 정의하여 사용하면 코드의 가독성을 높일 수 있다.

```
typedef void (*sigHandlerType)(int);
extern sigHandlerType signal(int signum, sigHandlerType handler);
```

[문제코드] 아래의 예제에서 getFunction() 함수에 대한 선언은 읽기도 어렵고 이해하기도 힘들다.

```
#include <stdio.h>

int (*getFunction(int))(void);
int getFruits();
int getGrains();

int main()
{
    int (*func)(void);
    func = getFunction(1);
    func();
    func = getFunction(2);
    func();

    getchar();
    return 0;
}
```

```
int (*getFunction(int type))(void)
{
    int (*func)(void);

    switch (type)
    {
        case 1:
            func = getFruits;
            break;
        case 2:
            func = getGrains;
            break;
    }

    return func;
}

int getFruits()
{
    printf("getFruits() 함수가 선택되었습니다.\n");
    return 1;
}

int getGrains()
{
    printf("getGrains() 함수가 선택되었습니다.\n");
    return 1;
}
```

[해결방법] typedef 를 이용하여 함수 포인터를 정의한 후, 사용

```
#include <stdio.h>

typedef int (*Func)(void);
Func getFunction(int);
int getFruits();
int getGrains();

int main()
{
    Func func;
    func = getFunction(1);
    func();
    func = getFunction(2);
    func();

    getchar();
    return 0;
}

Func getFunction(int type)
{
    Func func;
```

```

    switch (type)
    {
    case 1:
        func = getFruits;
        break;
    case 2:
        func = getGrains;
        break;
    }

    return func;
}

int getFruits()
{
    printf("getFruits() 함수가 선택되었습니다.\n");
    return 1;
}

int getGrains()
{
    printf("getGrains() 함수가 선택되었습니다.\n");
    return 1;
}

```

DCL-06. 프로그램 로직상의 고정적인 값을 나타낼 때에는 의미있는 심볼릭 상수를 사용하라.

- C언어에서는 리터럴 상수와 심볼릭 상수를 제공한다.
- 리터럴 상수는 소스코드의 가독성을 떨어뜨리며, 값을 변경할 경우, 비 효율적으로 수정되는 현상이 나타나므로 리터럴 상수를 직접 표기(하드코딩)하기 보다는 심볼릭 상수를 사용하는 것이 가독성을 높이고, 유지보수를 쉽게 할 수 있다.
- C언어에서의 3가지 심볼릭 상수
 - 매크로 상수
 - 열거형 상수
 - const 상수

const 상수

- const 상수는 const 상수가 선언된 블록 내에서만 사용이 가능하다.
- 컴파일러에 의해 type이 체크된다.
- 디버깅 도구로 디버깅이 가능하다.
- 함수 내부에서 사용하면 함수 호출시 마다 할당되고 초기화 되는 오버 헤드 발생한다.

const 상수는 컴파일 타임에서 정수형 상수가 필요한 곳에서는 사용할 수 없다.

[문제코드] const 상수는 구조체 멤버의 비트 크기로 사용할 수 없다.

```
#include <stdio.h>

const int bit_size = 3;

typedef struct _bitfield
{
    int a : 3;
    unsigned int b : bit_size;
} BitField;

int main()
{
    BitField bf = {-3, 7};

    printf("bf.a : %d\n", bf.a);
    printf("bf.b : %u\n", bf.b);

    getchar();
    return 0;
}
```

[해결방법] 매크로 상수나 열거형 상수를 사용

```
#include <stdio.h>

// #define bit_size 3
enum { bit_size=3 };

typedef struct _bitfield
{
    int a : 3;
    unsigned int b : bit_size;
} BitField;

int main()
{
    BitField bf = {-3, 7};

    printf("bf.a : %d\n", bf.a);
    printf("bf.b : %u\n", bf.b);

    getchar();
    return 0;
}
```

[문제코드] const 상수는 배열원소의 개수로 사용할 수 없다.

```
#include <stdio.h>

const int array_size = 3;

int main()
{
    int i;
    int arr[array_size] = {1, 2, 3};
    size_t size = sizeof(arr)/sizeof(arr[0]);

    for (i=0 ; i<size ; i++)
    {
        printf("arr[%d] : %d\n", i, arr[i]);
    }

    getchar();
    return 0;
}
```

[해결방법] 매크로 상수나 열거형 상수를 사용

```
#include <stdio.h>

#define array_size 3
enum { array_size=3 };

int main()
{
    int i;
    int arr[array_size] = {1, 2, 3};
    size_t size = sizeof(arr)/sizeof(arr[0]);

    for (i=0 ; i<size ; i++)
    {
        printf("arr[%d] : %d\n", i, arr[i]);
    }

    getchar();
    return 0;
}
```

[문제코드] const 상수는 switch문의 case 값으로 사용할 수 없다.

```
#include <stdio.h>
#include <stdio_ext.h>

const int strawberry = 3;
enum FRUITS { APPLE, CHERRY, ORANGE };

int main()
{
    int item;
    printf("상품번호를 입력하십시오 (0 ~ 3) : ");
    if (scanf("%d", &item) != 1){
        printf("입력오류!!!\n");
        return 1;
    }

    switch (item)
    {
        case APPLE:
            fputs("APPLE choice!", stdout); break;
        case CHERRY:
            fputs("CHERRY choice!", stdout); break;
        case ORANGE:
            fputs("ORANGE choice!", stdout); break;
        case strawberry:

```



```

        fputs("STRAWBERRY choice!", stdout); break;
    }

    __fpurge(stdin);
    getchar();
    return 0;
}

```

[해결방법]

```

#include <stdio.h>
#include <stdio_ext.h>

enum FRUITS { APPLE, CHERRY, ORANGE, strawberry };

int main()
{
    int item;
    printf("상품번호를 입력하시오 (0 ~ 3) : ");
    if (scanf("%d", &item) != 1){
        printf("입력오류!!!\n");
        return 1;
    }

    switch (item)
    {
        case APPLE:
            fputs("APPLE choice!", stdout);
            break;
        case CHERRY:
            fputs("CHERRY choice!", stdout);
            break;
        case ORANGE:
            fputs("ORANGE choice!", stdout);
            break;
        case strawberry:
            fputs("STRAWBERRY choice!", stdout);
            break;
    }

    __fpurge(stdin);
    getchar();
    return 0;
}

```

☞ 열거형 상수

- 열거형 상수는 int로 나타낼 수 있는 정수형 상수 표현식을 나타낼 때 사용한다.
- 별도의 메모리가 할당되지 않는다. (열거형 상수의 주소를 구할 수 없음)
- 열거형 상수는 type을 따로 지정할 수 없다. (항상 int형이다.)

```
#include <stdio.h>

enum { MAX=15 };

int main()
{
    int ary[MAX];    // 열거형 상수로 배열의 크기 지정 가능
    const int *p;
    p = ary;
    p = &MAX;        // 열거형 상수는 주소 계산이 불가능

    return 0;
}
```

☞ 객체형 매크로

```
#define identifier replacement-list
```

객체형 매크로 (Object-like Macro)는 전처리기(cpp)에 의해 프로그램 소스상의 매크로 이름(identifier)이 지시문의 나머지 부분(replacement-list)로 치환된다.

- 객체형 매크로는 메모리를 소비하지 않으므로 포인터로 가리킬 수 없다.
- 전처리기는 컴파일러가 심볼을 처리하기 전에 매크로 치환작업을 한다.
- 매크로는 타입 체크도 제공하지 않는다.

DCL-08. 상수 정의에서는 상수 간의 관계가 적절하게 나타나도록 정의하라.

한 정의가 다른 정의에 영향을 미친다면 둘 간의 관계를 인코딩(부호화)하고 각각을 따로 정의하지 마라. 또한 서로 관계가 없는 것들을 인코딩 하지마라.

[문제코드] OUT_STR_LEN은 항상 IN_STR_LEN보다 커야 한다면

```
enum { IN_STR_LEN=18, OUT_STR_LEN=20 };
```

[해결방법] 두 정의간의 관계를 명시적으로 나타내어 표현한다.

```
enum { IN_STR_LEN=18, OUT_STR_LEN=IN_STR_LEN+2 };
```

[문제코드] 서로에게 상관없는 관계가 성립되어 있다.

```
enum { ADULT_AGE = 18 };  
enum { ALCOHOL_AGE = ADULT_AGE + 3 };
```

[해결방법] 상관없는 관계는 정의에 포함하지 않는다.

```
enum { ADULT_AGE = 18 };  
enum { ALCOHOL_AGE = 21 };
```

DCL-15. 블록 범위를 넘어서까지 사용되지 않을 객체는 static으로 선언하라.

유형	설명
변수	선언된 블록에서만 접근이 가능하다.
함수	정의된 파일 내에서만 접근이 가능하다.

[문제코드] helper() 함수는 오직 util.c에서만 호출이 가능하여야만 한다.

[util.h 의 내용]

```
#ifndef UTIL_H_
#define UTIL_H_

#include<stdio.h>
void helper(int);

#endif
```

[util.c 의 내용]

```
void helper(int i)
{
    printf("이 함수는 오직 해당 파일에서만 호출이 가능하여야 합니다.\n");
}

void func()
{
    helper(3);
}
```

[main.c 의 내용]

```
#include <stdio.h>
#include "util.h"
int main()
{
    printf("helper()를 호출합니다.\n");
    helper(1);
    printf("func()를 호출합니다.\n");
    func();

    getchar();
    return 0;
}
```

[해결방법] helper() 함수를 static 키워드를 이용하여 정의함.

[util.h 의 내용]

```
#ifndef UTIL_H_
```

```
#define UTIL_H_

void helper(int);
#endif

[util.c 의 내용]
#include<stdio.h>

static void helper(int i)
{
    printf("이 함수는 오직 해당 파일에서만 호출이 가능하여야 합니다.\n");
}
void func()
{
    helper(3);
}

[main.c 의 내용]
#include <stdio.h>
#include "util.h"

int main()
{
    printf("helper()를 호출합니다.\n");
    helper(1); // ← 에러 발생
    printf("func()를 호출합니다.\n");
    func();

    getchar();
    return 0;
}
```

[예제] 생각해볼 문제 (static 변수는 외부에서 접근이 불가능 한가?)

```
#include <stdio.h>
#include <string.h>
int ary[5] = {10, 20, 30, 40, 50};

void sub()
{
    static int snum = 3;
    printf("snum의 값\t: %d\n", snum++);
    printf("snum의 주소\t: %p\n", &snum);
}

int main()
{
    ary[5]++;
}
```

```
    sub();  
    ary[5]++;  
    sub();  
  
    getchar();  
    return 0;  
}
```

DCL-33. 함수 인자에서 restrict로 지정된 소스 포인터와 목적 포인터가 동일한 객체를 참조하지 않게 하라.

restrict 지정자로 지정된 함수의 두 인자가 동일한 공간을 참조하는 경우에는 그 결과를 예측할 수 없다.

o restrict 키워드

포인터에만 사용가능한 지정자로서 컴파일러가 특정 유형의 코드를 최적화 할 수 있도록 허용함으로써 연산능력을 향상시킨다.

restrict로 지정된 포인터는 그 포인터가 포인터가 가리키는 공간에 접근할 수 있는 유일한 최초의 수단이라는 것을 나타낸다.

몇몇 C99 함수는 restrict 지정자를 사용한 매개변수를 갖는 함수들이 있다.

☞ 아래의 예제는 일부 컴파일러에서 문제를 발생시킬 수 있다.

[문제코드] ptr1과 ptr2는 같은 배열을 가리킨다.

```
#include <stdio.h>
#include <stdlib.h> // VS에서는 string.h 인클루드

int main()
{
    char str[] = "test string";
    char *ptr1 = str;
    char *ptr2 = ptr1 + 3;

    memcpy(ptr2, ptr1, 6);
    printf("ptr1 : %s\n", ptr1);
    printf("ptr2 : %s\n", ptr2);

    getchar();
    return 0;
}
```

[해결방법] memcpy() 함수 대신 memmove() 함수를 사용

```
#include <stdio.h>
#include <stdlib.h> // VS에서는 string.h 인클루드

int main()
{
    char str[] = "test string";
    char *ptr1 = str;
```

```
char *ptr2 = ptr1 + 3;

memmove(ptr2, ptr1, 6);
printf("ptr1 : %s\n", ptr1);
printf("ptr2 : %s\n", ptr2);

getchar();
return 0;
}
```

EXP-00. 연산자 우선순위를 나타내는 데 괄호를 사용하라.

[문제코드]

<code>x & 1 == 0</code>	◀ x의 최하위 비트를 테스트 하려하는가?
<code>int *p;</code> <code>int num = 3;</code> <code>p = &num;</code> <code>*p++;</code>	◀ p가 가리키는것을 증가하려 하는가?
<code>int num = 10;</code> <code>res = !num == 10;</code>	◀ num과 10을 비교 후, 논리 부정 하려 하는가?

[해결방법]

<code>(x & 1) == 0</code>	◀ x의 최하위 비트가 0인가를 테스트 한다.
<code>int *p;</code> <code>int num = 3;</code> <code>p = &num;</code> <code>(*p)++;</code>	◀ p가 가리키는 곳의 값을 사용한 후, 증가시킨다.
<code>int num = 10;</code> <code>res = !(num == 10);</code>	◀ num이 10과 같지 않은가를 테스트 한다.

EXP-02. 논리 연산자 AND와 OR의 단축 평가 방식을 알고 있어라.

논리 AND와 논리 OR 연산자는 첫 번째 피연산자로 평가가 완료되면 두번째 피 연산자는 평가하지 않는다.

연산자	좌측피연산자	우측피연산자
AND	거짓 인 경우	평가 안함
OR	참 인 경우	평가 안함

[문제코드]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>
#include <malloc.h>
#define BUF_SIZE 20

int getString(const char *, char **);

int main()
{
    int res;
    char *p;

    /* p는 NULL일수도 있고 아닐 수도 있다. */
    if (p || (p = (char *)malloc(BUF_SIZE)))
    {
        res = getString("이름입력", &p);
        if (res == 1)
        {
            printf("입력된 이름은 '%s'입니다.", p);
        }
        free(p);
        p = NULL;
    }
    else
    {
        printf("동적메모리 할당에러!!!\n");
        return 1;
    }

    __fpurge(stdin);
    getchar();
    return 0;
}

int getString(const char *q, char **p)
{
    char *nr = NULL;
```

```

printf("%s : ", q);
if (fgets(*p, BUF_SIZE, stdin) == NULL)
{
    return 0;
}
nr = strchr(*p, '\n');
if (nr != NULL) *nr = '\0';

return 1;
}

```

[해결방법]

```

#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>
#include <malloc.h>
#define BUF_SIZE 20

int getString(const char *, char **);

int main()
{
    int res;
    char *p;

    /* p가 NULL이 아니라면 초기화 하라. */

    p = (char *)malloc(BUF_SIZE);
    if (p == NULL)
    {
        printf("동적메모리 할당에러!!!\n");
        return 1;
    }

    res = getString("이름입력", &p);
    if (res == 1)
    {
        printf("입력된 이름은 '%s'입니다.", p);
    }
    free(p);
    p = NULL;

    __fpurge(stdin);
    getchar();
    return 0;
}

int getString(const char *q, char **p)
{
    char *nr = NULL;

    printf("%s : ", q);

```

```
if (fgets(*p, BUF_SIZE, stdin) == NULL)
{
    return 0;
}
nr = strchr(*p, '\n');
if (nr != NULL) *nr = '\0';

return 1;
}
```

EXP-03. 구조체의 크기가 구조체 멤버들 크기의 합이라고 가정하지 마라.

EXP-04. 구조체끼리 바이트 단위로 비교하지 마라.

[문제코드]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>

typedef struct _Person
{
    char name[30];
    int age;
} Person;

int main()
{
    Person *p1;

    p1 = (Person *)malloc(34);
    if (p1 == NULL)
    {
        printf("동적메모리 할당 실패!!!\n");
        return 1;
    }

    strcpy(p1->name, "KIHEE KIM");
    p1->age = 20;

    printf("NAME : %s\n", p1->name);
    printf("AGE : %d\n", p1->age);

    free(p1);
    p1 = NULL;

    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>

typedef struct _Person
{
    char name[30];
    int age;
} Person;

int main()
{
    Person *p1;

    p1 = (Person *)malloc(sizeof(Person));
    if (p1 == NULL)
    {
        printf("동적메모리 할당 실패!!!\n");
        return 1;
    }

    strcpy(p1->name, "KIHEE KIM");
    p1->age = 20;

    printf("NAME : %s\n", p1->name);
    printf("AGE : %d\n", p1->age);

    free(p1);
    p1 = NULL;

    getchar();
    return 0;
}
```

EXP-06. sizeof의 피연산자가 다른 부수 효과를 가지면 안된다.

[문제코드]

```
#include <stdio.h>

int main()
{
    int a;
    int b;

    a = 14;
    b = sizeof(a++);

    printf("a : %d\n", a);
    printf("b : %d\n", b);

    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>

int main()
{
    int a;
    int b;

    a = 14;
    a++;
    b = sizeof(a);

    printf("a : %d\n", a);
    printf("b : %d\n", b);

    getchar();
    return 0;
}
```

EXP-08. 포인터 연산이 정확하게 수행되고 있는지 보장하라.

○ 포인터 연산을 수행할 때 포인터에 더해지는 값은 자동적으로 포인터가 가리키는 데이터형으로 조정된다.

○ 포인터 연산의 특징

- 1) 주소상수 + 정수형상수(n) 은
주소상수 + (n * 주소상수에 해당하는 기억공간의 크기)
- 2) 주소상수 - 주소상수
주소에 해당하는 기억공간 간의 첨자(index)차이가 계산 됨

[문제코드]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>

#define BUFFER_SIZE 5
int getNumber(int *);

int main()
{
    int i;
    int p;
    int buffer[BUFFER_SIZE];
    int *bufptr = buffer;

    while (bufptr < (buffer + sizeof(buffer)))
    {
        if (getNumber(&p))
        {
            *bufptr++ = p;
        }
    }

    for (i=0 ; i<BUFFER_SIZE ; i++)
    {
        printf("buffer[%d] : %d\n", i, *(buffer+i));
    }

    __fpurge(stdin);
    getchar();
    return 0;
}
```

```
int getNumber(int *ptr)
{
    char tmp[10];
    char *nr, *br;

    *ptr = 0;
    memset(tmp, 0, sizeof(tmp));
    printf("3자리 정수입력 : ");
    while (1)
    {
        if (fgets(tmp, sizeof(tmp), stdin) == NULL)
        {
            return 0;
        }
        // fgets()함수로 받아들인 문자열에 개행문자가 있다고 가정하지
        // 마라.

        nr = strchr(tmp, '\n');
        if(nr!=NULL) *nr = '\0';
        else        fflush(stdin);
        if (strlen(tmp) == 0 || strlen(tmp) > 3)
        {
            return 0;
        }

        // 문자가 없거나 중간에 숫자가 아닌 문자가 존재하는가?
        *ptr = (int)strtol(tmp, &br, 10);
        if (br == tmp || *br != '\0')
        {
            return 0;
        }

        break;
    }
    return 1;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>

#define BUFFER_SIZE 5
int getNumber(int *);

int main()
{
    int i;
    int p;
    int buffer[BUFFER_SIZE];
    int *bufptr = buffer;
```

```

while (bufptr < (buffer + BUFFER_SIZE))
{
    if (getNumber(&p))
    {
        *bufptr++ = p;
    }
}

for (i=0 ; i<BUFFER_SIZE ; i++)
{
    printf("buffer[%d] : %d\n", i, *(buffer+i));
}

__fpurge(stdin);
getchar();
return 0;
}

int getNumber(int *ptr)
{
    char tmp[10];
    char *nr, *br;

    *ptr = 0;
    memset(tmp, 0, sizeof(tmp));
    printf("3자리 정수입력 : ");
    while (1)
    {
        if (fgets(tmp, sizeof(tmp), stdin) == NULL)
        {
            return 0;
        }
        // fgets()함수로 받아들이는 문자열에 개행문자가 있다고 가정하지 마라.
        nr = strchr(tmp, '\n');
        if(nr!=NULL) *nr = '\0';
        else
            fflush(stdin);

        if (strlen(tmp) == 0 || strlen(tmp) > 3)
        {
            return 0;
        }

        // 문자가 없거나 중간에 숫자가 아닌 문자가 존재하는가?
        *ptr = (int)strtol(tmp, &br, 10);
        if (br == tmp || *br != '\0')
        {
            return 0;
        }
        break;
    }
    return 1;
}

```

EXP-10. 하위 표현식의 평가 순서나 부수효과가 발생할 수 있는 영역의 순서에 의존하지 마라.

하위 표현식의 평가나 부수효과가 발생하는 순서가 지정되어 있지 않은 경우

- 함수에 주어진 인자가 평가되는 순서
- 할당문에서 피연산자가 평가되는 순서
- 초기화 표현식에서 나열된 객체들이 부수 효과를 갖는 순서
(객체의 초기화 순서가 나열 순서와 같을 필요가 없다.)

다음 코드의 실행결과, extern변수 g에 1이 할당될 가능성과 2가 할당될 가능성은 반반이다.

[문제코드]

```
#include <stdio.h>

int g;

int f(int i)
{
    g = i;
    return i;
}

int main()
{
    int x = f(1) + f(2);

    printf("extern g의 값 : %d\n", g);

    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>

int g;

int f(int i)
{
    g = i;
    return i;
}
```

```
}  
  
int main()  
{  
    int x = f(1);  
    x += f(2);  
  
    printf("extern g의 값 : %d\n", g);  
  
    getchar();  
    return 0;  
}
```

EXP-11. 호환되지 않는 타입들에는 연산자를 적용하지 마라.

비트필드 구조에 멤버가 할당되는 순서는 컴파일러마다 차이가 있다.

[문제코드]

```
#include <stdio.h>

struct bf
{
    unsigned int m1 : 8;
    unsigned int m2 : 8;
    unsigned int m3 : 8;
    unsigned int m4 : 8;
};

int main()
{
    struct bf data;
    unsigned char *ptr;

    data.m1 = 0;
    data.m2 = 0;
    data.m3 = 0;
    data.m4 = 0;
    ptr = (unsigned char *) &data;
    (*ptr)++;

    printf("data.m1 : %u\n", data.m1);
    printf("data.m2 : %u\n", data.m2);
    printf("data.m3 : %u\n", data.m3);
    printf("data.m4 : %u\n", data.m4);

    getchar();
    return 0;
}
```

[해결방법] 수정할 필드를 명백히 지정한다.

```
#include <stdio.h>

struct bf
{
    unsigned int m1 : 8;
    unsigned int m2 : 8;
    unsigned int m3 : 8;
    unsigned int m4 : 8;
};

int main()
```

```
{
    struct bf data;

    data.m1 = 0;
    data.m2 = 0;
    data.m3 = 0;
    data.m4 = 0;
    data.m1++;          // 값을 변경할 멤버를 명확히 지정하라.

    printf("data.m1 : %u\n", data.m1);
    printf("data.m2 : %u\n", data.m2);
    printf("data.m3 : %u\n", data.m3);
    printf("data.m4 : %u\n", data.m4);

    getchar();
    return 0;
}
```

EXP-33. 초기화 되지 않은 메모리를 참조하지 마라.

- auto 변수는 초기화 되기 전에는 garbage 데이터가 들어있다.
- malloc()함수로 동적 할당된 메모리 공간에도 정해지지 않은 값들이 들어있기 때문에 초기화 하기 전에 사용하면 예상하지 못한 결과를 초래한다.

☞ 컴파일러는 초기화되지 않은 변수의 주소가 함수에 전달될 때, 변수가 함수내에서 초기화된다고 가정한다. 이렇게 컴파일러는 변수 초기화가 실패해도 진단하지 못하므로 프로그래머가 정확성을 위해 부가적으로 점검하여야 한다.

[문제코드]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>
#define BUF_NAME_SIZE 100

void getName(char *);

int main()
{
    char *name;
    getName(name);

    printf("입력된 이름 : %s\n", name);

    __fpurge(stdin);
    getchar();
    return 0;
}

void getName(char *name)
{
    char *nr = NULL;
    name = (char *)malloc(BUF_NAME_SIZE);
    if (name == NULL)
    {
        printf("동적메모리 할당실패!!!\n");
        return;
    }

    printf("이름입력 : ");
    fgets(name, BUF_NAME_SIZE, stdin);
    nr = strchr(name, '\n');
    if (nr != NULL) *nr = '\0';

    return;
}
```


[해결방법 1]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>
#define BUF_NAME_SIZE 100

char * getName();

int main()
{
    char *name;
    name = getName();

    printf("입력된 이름 : %s\n", name);

    free(name);
    __fpurge(stdin);
    getchar();
    return 0;
}

char * getName()
{
    char *name = NULL;
    char *nr = NULL;

    name = (char *)malloc(BUF_NAME_SIZE);
    if (name == NULL)
    {
        printf("동적메모리 할당실패!!!\n");
        return name;
    }

    printf("이름입력 : ");
    fgets(name, BUF_NAME_SIZE, stdin);
    nr = strchr(name, '\n');
    if (nr != NULL) *nr = '\0';

    return name;
}
```

[해결방법 2]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <malloc.h>
#include <string.h>
#define BUF_NAME_SIZE 100

void getName(char **);

int main()
{
    char *name;
    getName(&name);

    printf("입력된 이름 : %s\n", name);

    free(name);

    __fpurge(stdin);
    getchar();
    return 0;
}

void getName(char **name)
{
    char *nr = NULL;
    *name = (char *)malloc(BUF_NAME_SIZE);
    if (*name == NULL)
    {
        printf("동적메모리 할당실패!!!\n");
        return;
    }

    printf("이름입력 : ");
    fgets(*name, BUF_NAME_SIZE, stdin);
    nr = strchr(*name, '\n');
    if (nr != NULL) *nr = '\0';

    return;
}
```

INT-00. 플랫폼에 따른 데이터 모델을 이해하고 있어라.

데이터 모델은 표준 데이터 타입에 대한 할당되는 크기를 정의한다. 사용하는 플랫폼에 따른 데이터 모델을 이해하는 일은 중요하다.

타입	iAPX68	IA32	IA64	SPARK64	ARM32	Alpha	64bit POSIX	ILP64
char	8	8	8	8	8	8	8	8
short	16	16	16	16	16	16	16	16
int	16	32	32	32	32	32	32	64
long	32	32	32	64	32	64	64	64
long long	N/A	64	64	64	64	64	64	64
포인터	16/32	32	64	64	32	64	64	64

☞ `limits.h` 파일에는 표준 정수 타입의 범위를 결정하는 매크로가 지정되어 있다.

`UINT_MAX` : `unsigned int`형이 가질 수 있는 최대 값

`LONG_MIN` : `long int`가 가질 수 있는 최소 값

☞ `stdint.h` 파일에는 데이터 모델에 종속되지 않고 사용할 수 있는 특정 크기로 제한된 타입들이 있다.

`int_least32_t` : 플랫폼에서 지원하는 최소 32비트 이상의 `signed`형 정수타입

`uint_fast16_t` : 최소 16비트 이상을 갖는 `unsigned` 정수 타입

[문제코드] `unsigned int`값 두 개를 곱할 때 64비트 리눅스 시스템에서는 정상 동작 하지만 윈도우 시스템에서는 문제가 발생한다.

```
#include <stdio.h>

int main()
{
    unsigned int a, b;
    unsigned long c;

    a = 1000000000;
    b = 100;
    c = (unsigned long)a * b;

    printf("c = %lu\n", c);

    getchar();
    return 0;
}
```

[해결방법] 결과를 보존할 수 있는 큰 unsigned int 타입을 사용

```
#include <stdio.h>
#include <limits.h>
#include <stdint.h>

int main()
{
    unsigned int a, b;
    uintmax_t c;

    a = 1000000000; // 10억
    b = 100;
    c = (uintmax_t)a * b;

    printf("c = %llu\n", c);

    getchar();
    return 0;
}
```

INT-01. 배열의 크기를 나타내는 값이나 루프 카운터로 사용되어지는 변수의 데이터 형은 unsigned 정수형 또는 size_t 형을 사용하라.

[생각해볼 코드]

```
#include<stdio.h>

int main()
{
    char i;

    for(i=0; i<100; i--)
    {
        printf("i=%d\n", i);
    }
    getchar();

    for(i=0; i<200; i++)
    {
        printf("i=%d\n", i);
    }
    getchar();

    return 0;
}
```

[생각해볼 코드]

```
#include<stdio.h>

int main()
{
    char x = 127;
    char y = x + 1;
    printf("x = %d y = %d\n", x, y);

    x = - 128;
    y = x - 1;
    printf("x = %d y = %d\n", x, y);

    getchar();
    return 0;
}
```

size_t 타입은 sizeof연산의 결과로 얻어지는 unsigned 정수 타입이다. size_t의 한계값은 SIZE_MAX 매크로로 지정되어 있다.

[문제코드] copy()함수의 루프카운터로 사용된 변수 i는 signed int 형으로 n>INT_MAX 인 값에 대해서 오버플로우가 발생한다.

```
#include <stdio.h>
#include <stdlib.h>

char *copy(const char *str, size_t n)
{
    int i;
    char *p;
    p = (char *)malloc(n * sizeof(char));
    if (p == NULL)
    {
        printf("동적할당오류!!!\n");
        return NULL;
    }

    for (i=0 ; i<n ; i++)
    {
        p[i] = *str++;
    }
    return p;
}

int main()
{
    char *message;
    char string[] = "문자열을 복사합니다.";

    if ((message = copy(string, sizeof(string))) != NULL)
    {
        printf("원본 : %s\n", string);
        printf("사본 : %s\n", message);
    }
    else
    {
        printf("문자열 복사 실패!!!\n");
        getchar();
        return 1;
    }

    getchar();
    return 0;
}
```

[해결방법] i를 size_t 타입으로 수정한다.

```
#include <stdio.h>
#include <stdlib.h>

char *copy(const char *str, size_t n)
{
    size_t i;
    char *p;
    p = (char *)malloc(n * sizeof(char));
    if (p == NULL)
    {
        printf("동적할당오류!!!\n");
        return NULL;
    }

    for (i=0 ; i<n ; i++)
    {
        p[i] = *str++;
    }
    return p;
}

int main()
{
    char *message;
    char string[] = "문자열을 복사합니다.";

    if ((message = copy(string, sizeof(string))) != NULL)
    {
        printf("원본 : %s\n", string);
        printf("사본 : %s\n", message);
    }
    else
    {
        printf("문자열 복사 실패!!!\n");
        getchar();
        return 1;
    }

    getchar();
    return 0;
}
```

INT-02. 정수 변환 규칙을 이해하라.

▶ 정수변환

- 캐스팅을 통한 명시적인 변환
- 연산식에 의한 묵시적인 변환

▶ 정수변환 규칙

1. 정수의 승계
2. 정수 변환 순위
3. 일반적인 산술변환

1) 정수의 승계

int보다 작은 정수형은 연산이 수행될 때 int형나 unsigned int형으로 변환되어 연산된다.
정수의 승계는 연산의 중간에 사용되는 값의 오버플로우를 예방하기 위해서 수행된다.

```
#include <stdio.h>

int main()
{
    signed char result, c1, c2, c3;

    c1 = 100;
    c2 = 3;
    c3 = 4;

    /* 300 / 4 *의 결과 값이
    result에 signed char형으로 변환되어 저장된다. */
    result = c1 * c2 / c3;
    printf("result : %d\n", result);

    getchar();
    return 0;
}
```

2) 정수 변환 순위

비트수가 많은 자료형이 높은 순위를 갖는다. 정수 변환 순위는 일반적인 산술변환에서 각기 다른 정수 타입에 대한 연산이 수행될 때, 어떤 변환이 필요한지를 결정하는데 사용한다.

- 각기 다른 두 signed 정수형은 순위가 다르다.
- signed 정수형의 변환 순위는 자신보다 정밀도가 작은 다른 signed 정수형보다 순위가 높다.

`long long int > long int > int > short > char`

☞ 모든 unsigned 정수형 순위는 일치하는 signed 정수형의 순위와 같다.

3) 일반적인 산술변환

두 피연산자의 자료형을 일치시켜야 하는 경우에 적용하는 일종의 규칙

- 이항 연산 시 두 피연산자가 같은 자료형으로 변환 된다.
- 조건 연산자 (?:)의 두번째와 세번째 피연산자는 같은 자료형으로 변환된다.
 - 정수의 승계가 먼저 진행된다.
 - 두 개의 피연산자가 같은 자료형이면 변환하지 않는다.
 - unsigned 정수형 피연산자의 자료형이 다른 피연산자의 자료형의 순위보다 크거나 같거나 큰 경우, signed 정수형의 피연산자의 데이터형이 unsigned 정수형의 피연산자 데이터형으로 변환된다.
 - signed 정수형 피연산자의 자료형이 다른 피연산자의 자료형보다 순위보다 큰 경우, signed 정수형 피연산자의 자료형이 다른 정수형 피연산자의 값을 표현할 수 있다면 signed 정수형 피연산자의 자료형으로 변환된다. 반면 signed 정수형 피연산자의 자료형이 다른 정수형 피연산자의 값을 표현 할 수 없다면 signed 정수형 피연산자와 일치하는 unsigned 정수형으로 두 피연산자가 모두 변환된다.

[정수변환규칙의 예]

char : 8bit / int : 32bit / long long 64bit 시스템에서

```
signed char sc = SCHAR_MAX;  
unsigned char uc = UCHAR_MAX;  
signed long long all = sc + uc;
```

(1) 정수의 승계에 의해 sc, uc 모두 int로 변환된다.

```
signed long long = int;
```

(2) int의 결과에서 sign부분만 64비트 영역으로 확장된다.

※ 안전하게 수행 됨

[정수 변환의 예]

char : 32bit / int : 32bit / long long : 64bit 시스템에서

```
signed char sc = SCHAR_MAX;  
unsigned char uc = UCHAR_MAX;  
signed long long all = sc + uc;
```

(1) sc는 int형으로, uc는 unsigned int형으로 변환된다.

```
signed long long = signed int + unsigned int
```

(2) signed int가 unsigned int형으로 변환된다.

```
signed long long = unsigned int + unsigned int
```

※ 이때 unsigned int + unsigned int 에서 오버플로우가 발생한다.

(3) 결과값이 signed long long 형으로 변환되어 저장 된다.

[생각해볼 예제]

```
#include <stdio.h>

int main()
{
    int x = -1;
    unsigned int y = 1;

    if (x > y)
    {
        printf("True\n");
    }
    else
    {
        printf("False\n");
    }

    getchar();
    return 0;
}
```

[생각해볼 예제]

```
#include <stdio.h>

int main()
{
    char x = -1;
    unsigned char y = 1;

    if (x > y)
    {
        printf("True\n");
    }
    else
    {
        printf("False\n");
    }

    getchar();
    return 0;
}
```

[문제코드] 서로 다른 타입의 연산을 수행할 때 주의해야 한다.

```
#include <stdio.h>

int main()
{
    int si = -1;
    unsigned int ui = 1;

    printf("%d\n", si < ui);

    getchar();
    return 0;
}
```

[문제해결] signed int 값을 사용해 비교하도록 강제형변환을 한다.

```
#include <stdio.h>

int main()
{
    int si = -1;
    unsigned int ui = 1;

    printf("%d\n", si < (int)ui);

    getchar();
    return 0;
}
```

INT-04. 불분명한 소스에서 얻어지는 정수의 값은 제한을 강제하라.

신뢰할 수 없는 소스로 부터 얻어지는 정수 값은 식별할 수 있는 상한값과 하한값이 있는지 확인하기 위해 반드시 평가되어야 한다.

아래 코드에서 table의 길이를 결정하는데 쓰이는 length의 값을 신뢰할 수 없다면 malloc() 호출 실패등의 문제가 발생할 수 있으므로 length의 적용가능한 범위를 평가하여야 한다.

[문제코드]

```
#include <stdio.h>
#include <stdlib.h>

enum { MAX_TABLE_LENGTH = 256 };

int create_table(size_t);
void myflush();

int main()
{
    int length;

    while (1)
    {
        printf("문자열의 수 : ");
        if (scanf("%d", &length) != 1 || length < 0)
        {
            myflush();
            printf("입력오류");
            continue;
        }

        create_table(length);
        break;
    }

    myflush();
    getchar();
    return 0;
}

int create_table(size_t length)
{
    size_t table_length;
    char **table;

    table_length = length * sizeof(char *);
    table = (char **)malloc(table_length);
    if (table == NULL)
    {
        printf("동적메모리할당실패!!\n");
        return 0;
    }
}
```

```

    }

    printf("동적메모리할당성공!!!\n");
    /* table 을 사용하는 코드 */
    free(table);
    return 1;
}

void myflush()
{
    while (getchar() != '\n');
}

```

[해결방법]

```

#include <stdio.h>
#include <stdlib.h>

enum { MAX_TABLE_LENGTH = 256 };

int create_table(size_t);
void myflush();

int main()
{
    int length;

    while (1)
    {
        printf("문자열을 수 : ");
        if (scanf("%d", &length) != 1 || length < 0)
        {
            myflush();
            printf("입력오류");
            continue;
        }

        create_table(length);
        break;
    }

    myflush();
    getchar();
    return 0;
}

int create_table(size_t length)
{
    size_t table_length;
    char **table;

    if (length == 0 || length > MAX_TABLE_LENGTH)

```

```

{
    printf("유효하지 않은 수 입니다.\n");
    return 0;
}

table_length = length * sizeof(char *);
table = (char **)malloc(table_length);
if (table == NULL)
{
    printf("동적메모리할당실패!!!\n");
    return 0;
}

printf("동적메모리할당성공!!!\n");
/* table 을 사용하는 코드 */
free(table);
return 1;
}

void myflush()
{
    while (getchar() != '\n');
}

```

INT-05. 모든 가능한 입력을 처리할 수 없다면 문자 데이터 변환을 위해 입력함수를 사용하지 마라.

문자열을 읽어 정수나 부동소수점 수로 변환하여 저장하는 함수는 인자 타입으로 표현 불가능한 숫자를 담을 수 없으므로 사용하지 않는 것이 좋다.

[문제코드] scanf()함수는 입력 문자열을 정수로 변환할 수 없는 경우, 정의되지 않은 행동을 유발한다.

```
#include <stdio.h>
#include <stdio_ext.h>

int main()
{
    long s1;
    int res;

    printf("long형 정수입력 : ");
    res = scanf("%ld", &s1);
    if (res != 1)
    {
        printf("데이터 입력 오류가 발생하였습니다.\n");
    }

    printf("입력된 데이터 : %ld\n", s1);
    printf("scanf()함수 실행결과 : %d\n", res);

    __fpurge(stdin);
    getchar();
    return 0;
}
```

[해결방법] 입력 문자열은 `fgets()` 함수를 사용하여 입력하고, 문자열을 정수로 변환하기 위해 `strtol()` 함수를 사용한다. `strtol()` 함수는 입력 값이 long영역에서 유효한지를 점검하는 에러 체크 메커니즘을 제공한다.

```
#include <stdio.h>
#include <stdio_ext.h>
#include <stdlib.h>
#include <limits.h>
#include <errno.h>

int main()
{
    long s1;
    char buff[25];
    char *endptr;

    fgets(buff, sizeof(buff), stdin);

    errno = 0;
    s1 = strtol(buff, &endptr, 10);

    if (errno == ERANGE)
    {
        printf("long형 범위 밖의 값이 입력되었습니다.\n");
    }
    else if (endptr == buff && *endptr == '\n') // 그냥 엔터만 입력 시
    {
        printf("데이터가 입력되지 않았습니다.\n");
    }
    else if (*endptr != '\n' && *endptr != '\0')
    {
        printf("숫자로 변환할 수 없는 문자가 포함되어 있습니다.\n");
    }
    else
    {
        printf("입력된 데이터 : %ld\n", s1);
    }

    __fpurge(stdin);
    getchar();
    return 0;
}
```

INT-06. 문자열 토큰을 정수로 변환할 때에는 strtol()이나 관련 함수를 사용하라.

문자열 토큰을 정수로 변환할 때에는 strtol()이나 관련 함수를 사용하라. 아래의 함수들은 범위 체크 등 신뢰성 있는 에러 처리를 제공한다.

strtol(), strtoll(), strtoul(), strtoull(), strtod(), strtodf()

위의 함수들은 다음과 같은 기능을 제공한다.

- 에러 발생 시 `errno`를 설정한다.
- 문자열에 정수 값이 없을 경우, 리턴되는 0과 실제 0이 입력된 경우를 변환 에러 발생 위치의 확인을 통해 확인이 가능하다.

아래의 함수들은 부적절한 입력에 대한 에러를 출력해주는 메커니즘이 부실하므로 사용하지 않는것이 좋다.

atoi(), atol(), atof(), sscanf()

INT-07. 숫자를 저장할 char형 변수에는 명시적으로 signed나 unsigned를 지정하라.

char, signed char, unsigned char를 통틀어서 문자형(character type)이라고 한다. char형 변수에 저장되는 값은 문자가 아닌 해당 문자의 ASCII 코드값이 저장된다.

[문제코드] char형 변수 c는 signed형일 수도 있고, unsigned형일 수도 있다.

```
#include <stdio.h>

int main()
{
    char c = 200; // -56 or 200으로 인식
    int i = 1000;

    printf("i/c = %d\n", i/c);

    getchar();
    return 0;
}
```

[해결방법] signed 또는 unsigned형임을 명확히 명시한다.

```
#include <stdio.h>

int main()
{
    unsigned char c = 200; // 200으로 인식
    int i = 1000;

    printf("i/c = %d\n", i/c);

    getchar();
    return 0;
}
```

INT-09. 열거형 상수가 유일한 값으로 매핑되도록 보장하라.

C의 열거형은 정수형으로 매핑된다. 일반적으로 각 열거형 상수는 고유한 개별의 값(매핑하지 않는 경우 첫번째 멤버의 값이 0)으로 매핑된다고 생각하지만, 열거형 타입 멤버들이 서로 같은 값을 갖는 명확하지 않은 예러가 종종 만들어지기도 한다.

아래의 코드에서는 열거형 멤버의 값이 명시적으로 매핑되고 있으나 값이 중복되는 문제가 발생한다. 이로 인해 yellow와 indigo가 green과 violet가 같은 상수값을 지니게 된다. 그러므로 yellow와 indigo 그리고 green과 violet는 사용함에 있어서 제약이 따르게 된다.

switch ~ case 구문에서의 case의 상수값은 중복될 수 없다.

```
enum { red=4, orange, yellow, green, indigo=6, violet };
```

열거형 타입의 선언은 반드시 아래의 내용을 따르도록 하라.

-
-
1. 명시적인 선언을 하지 않는다.

```
enum { red, orange, yellow, green, indigo, violet };
```

2. 첫 번째 멤버에 대해서만 값을 지정한다.

```
enum { red=4, orange, yellow, green, indigo, violet };
```

3. 모든 멤버에 대하여 명시적으로 값을 지정한다.

```
enum { red=4, orange=5, yellow=6, green=7, indigo=6, violet=7 };
```

INT-10. % 연산자를 쓸 때 나머지가 양수라고 가정하지 마라.

- C89에서는 음수 피연산자에 대한 나머지 연산의 의미가 구현마다 다르게 정의된다.
- 여러 플랫폼의 다양한 컴파일 환경을 가진 경우, 개발자는 % 연산자의 동작을 표준에 의지할 수는 없다.
- C99에서 결과의 부호는 피제수(좌측의 피연산자)의 부호를 따른다.

[문제코드] 모듈로 형식의 수식의 결과로 배열의 index를 지정하는데 있어서 피제수가 int 형이므로 항상 양수임을 보장할 수가 없다.

```
#include <stdio.h>
#include <stdio_ext.h>

int insert(int, int *, int, int);

int main()
{
    int i;
    int index;
    int value;
    int ary[5] = {0,};
    int size = sizeof(ary)/sizeof(ary[0]);

    while (1)
    {
        printf("입력할 위치 : ");
        __fpurge(stdin);
        if (scanf("%d", &index) != 1)
        {
            printf("데이터입력오류!!!\n");
            return 1;
        }

        printf("입력할 데이터 : ");
        __fpurge(stdin);
        if (scanf("%d", &value) != 1)
        {
            printf("데이터입력오류!!!\n");
            return 1;
        }

        index = insert(index, ary, size, value);
        printf("%d번째 방의 값을 초기화 하였습니다.\n", index);
        for (i=0 ; i<size ; i++)
        {
            printf("ary[%d] : %d\t", i, ary[i]);
        }
    }
}
```

```

        }
        printf("\n");
    }

    __fpurge(stdin);
    getchar();
    return 0;
}

int insert(int index, int *list, int size, int value)
{
    if (size != 0)
    {
        index = index % size;
        list[index] = value;
        return index;
    }
    else
    {
        return -1;
    }
}

```

[해결방법]

```

#include <stdio.h>
#include <stdio_ext.h>

int insert(size_t, int *, int, int);

int main()
{
    int i;
    int index;
    int value;
    int ary[5] = {0,};
    int size = sizeof(ary)/sizeof(ary[0]);

    while (1)
    {
        printf("입력할 위치 : ");
        __fpurge(stdin);
        if (scanf("%d", &index) != 1)
        {
            printf("데이터입력오류!!!\n");
            return 1;
        }

        printf("입력할 데이터 : ");
        __fpurge(stdin);
        if (scanf("%d", &value) != 1)
        {

```

```

        printf("데이터입력오류!!!\n");
        return 1;
    }

    index = insert(index, ary, size, value);
    printf("%d번째 방의 값을 초기화 하였습니다.\n", index);
    for (i=0 ; i<size ; i++)
    {
        printf("ary[%d] : %d\t", i, ary[i]);
    }
    printf("\n");
}

__fpurge(stdin);
getchar();
return 0;
}

int insert(size_t index, int *list, int size, int value)
{
    if (size != 0)
    {
        index = index % size;
        list[index] = value;
        return index;
    }
    else
    {
        return -1;
    }
}

```

INT-12. 표현식에서 signed, unsigned 표시가 없는 int비트 필드의 타입을 가정하지 마라.

[문제코드] 비트필드에서 int형 비트필드를 지정하면 signed int형인지 unsigned int형인지 모호해 진다.

```
#include <stdio.h>
#include <stdio_ext.h>

struct {
    int a : 8;
} bits = {255};

int main()
{
    printf("bits.a : %d\n", bits.a);

    __fpurge(stdin);
    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdio_ext.h>

struct {
    unsigned int a : 8;
} bits = {255};

int main()
{
    printf("bits.a : %d\n", bits.a);

    __fpurge(stdin);
    getchar();
    return 0;
}
```

INT-13. 비트 연산자는 unsigned 피연산자에만 사용하라.

비트연산자(~, >>, <<, &, |, ^)는 signed 정수에 대한 비트연산이 구현마다 다르게 정의되어 있으므로 unsigned 정수 피연산자에 대해서만 사용하도록 하자.

[문제코드] signed 정수형에 대한 >> 연산은 부호값이 빈 공간에 채워지게 된다.

```
#include <stdio.h>
#include <stdio_ext.h>
#include <limits.h>

void charPrinter(char);

int main()
{
    char ch = CHAR_MIN;

    charPrinter(ch);
    ch >>= 1;
    charPrinter(ch);

    __fpurge(stdin);
    getchar();
    return 0;
}

void charPrinter(char v)
{
    unsigned char op = 1;
    op <<= sizeof(char) * CHAR_BIT - 1;

    printf("%10d : ", v);
    while (op > 0)
    {
        if (v & op) printf("1");
        else printf("0");
        op >>= 1;
    }
    printf("\n");
}
```

[해결방법]

```
#include <stdio.h>
#include <stdio_ext.h>
#include <limits.h>

void charPrinter(char);

int main()
```

```
{
    unsigned char ch = CHAR_MIN;

    charPrinter(ch);
    ch >>= 1;
    charPrinter(ch);

    __fpurge(stdin);
    getchar();
    return 0;
}

void charPrinter(char v)
{
    unsigned char op = 1;
    op <<= sizeof(char) * CHAR_BIT - 1;

    printf("%10d : ", v);
    while (op > 0)
    {
        if (v & op) printf("1");
        else printf("0");
        op >>= 1;
    }
    printf("\n");
}
```

INT-30. unsigned 정수 연산이 래핑되지 않도록 주의하라.

○ unsigned 피 연산자를 사용한 계산은 결코 오버플로우가 발생하지 않는다. 연산 결과 값이 저장될 정수 타입으로 표현할 수 없는 경우, 나머지(%)연산으로 값을 줄여(wrap around) 표현하기 때문이다. 이것을 정수 래핑(wrapping) 현상이라고 한다.

○ 신뢰할 수 없는 소스로부터 얻어진 정수 값이 아래와 같은 곳에 사용된다면 절대 래핑을 허용하여서는 안된다.

- 배열의 인덱스
- 포인터 연산의 일부
- 루프 카운터
- 메모리 할당 함수의 인자
- 그 밖의 보안에 민감한 코드

[문제코드] unsigned 피 연산자 끼리의 덧셈 과정에서 unsigned 정수 래핑이 발생한다.

```
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int ui1, ui2, sum = 0;

    ui1 = UINT_MAX;
    ui2 = INT_MAX;

    sum = ui1 + ui2;

    printf("sum : %ld\n", sum);

    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int ui1, ui2, sum = 0;

    ui1 = UINT_MAX;
    ui2 = INT_MAX;

    if (UINT_MAX - ui1 < ui2)
    {
        printf("래핑이 발생합니다.!!!\n");
        /* 래핑관련에러처리 */
    }
    else
    {
        sum = ui1 + ui2;
    }

    printf("sum : %ld\n", sum);

    getchar();
    return 0;
}
```

INT-34. 피연산자의 비트보다 더 많은 비트를 시프트하지 마라.

아래의 예제에서는 좌측 피연산자의 비트수 보다 더 많이 쉬프트를 하고 있다.
그러므로 아래 예제의 쉬프트 연산은 다음과 같이 수행된다.

```
result = ui1 << ui2;
▶ result = ui1 << (ui2 % ui1의비트수);
```

[문제코드] 피연산자의 비트수 보다 더 많은 비트를 shift하려고 한다.

```
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int i, ui1, ui2, uresult = 0;

    ui1 = 1;
    ui2 = 35; // shift 회수
    for(i=0; i<ui2; i++)
    {
        uresult = ui1 << i;
        printf("i = %u => uresult : %u\n", i, uresult);
    }

    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <limits.h>

#define INT_BIT CHAR_BIT*sizeof(unsigned int)

int main()
{
    unsigned int i, ui1, ui2, uresult = 0;

    ui1 = 1;
    ui2 = 35; // shift 회수

    for(i=0; i<ui2; i++)
    {
        if(i >= INT_BIT) {break;}
        uresult = ui1 << i;
        printf("i = %u => uresult : %u\n", i, uresult);
    }
}
```

```
    getchar();
    return 0;
}
```

[문제코드] unsigned 피연산자 끼리의 left shift 연산과정에서 unsigned 정수 래핑이 발생

```
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int ui1, ui2, uresult = 0;

    ui1 = 256;
    ui2 = 28;
    uresult = ui1 << ui2;
    printf("uresult : %u\n", uresult);

    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int ui1, ui2, uresult = 0;

    ui1 = 256;
    ui2 = 28;

    if (ui1 > (UINT_MAX >> ui2))
    {
        printf("래핑이 발생합니다.\n");
        /* 래핑에 대한 에러 처리 */
    }
    else
    {
        uresult = ui1 << ui2;
    }

    printf("uresult : %u\n", uresult);
    getchar();
}
```

```
    return 0;  
}
```

제6장 부동소수점 (FLP)

FLP-00. 부동소수점의 저장방식을 이해하라.

- IEEE754 표준에서 정의하고 있는 방식이 가장 일반적인 부동소수점 방식이다.
- 그러나 IBM의 부동소수점 표기방식도 사용되는데, 이러한 시스템들은 정밀도와 표현가능한 값의 범위가 각각 다르다.
- 즉, 시스템별로 동일한 부동소수점의 구현사항이 보장되지 않기 때문에 정밀도나 범위에 대해 어떤 가정도 하여서는 안된다.

[예제] 부동소수점의 정확도

```
#include <stdio.h>

int main()
{
    float f = 1.0f/3.0f;
    printf("float is %.40f\n", f);
    getchar();
    return 0;
}
```

[결과]

gcc에서의 결과

```
float is 0.3333333432674407958984375000000000000000
```

Dev-C에서의 결과

```
float is 0.3333333432674408000000000000000000000000
```


☞ 부동소수점의 저장방식 (IEEE754 표준)

① 단정도 부동소수 (float : 32bit)

sign bit	지수부	가수부(유효숫자부)
1bit	8bit	23bit

② 배정도 부동소수 (double : 64bit)

sign bit	지수부	가수부(유효숫자부)
1bit	11bit	52bit

● sign bit : 가수부(유효숫자부)의 부호 (양수 : 0, 음수 : 1)

● 지수부 저장방식 : 지수값 + Bias값

Bias값은 지수부가 표현할 수 있는 최대 크기값을 2로 나눈 값

단정도 : (지수부 8bit에 저장할 수 있는 최대값) / 2 이므로 127

배정도 : (지수부 11bit에 저장할 수 있는 최대값) / 2 이므로 1023

● 가수부 저장방식 : 2진수로 표현된 숫자를 정규화 한 후, 소수점 이하 자리만을 저장함

정규화란? 2진수로 변환시킨 실수의 소수점의 위치를 앞에 1이 위치할 때 까지 옮기는 것을 말한다. 이때 소수점이 이동한 횟수가 지수가 되며, 소수점이 좌측으로 이동하였다면 양의 값, 우측으로 이동하였다면 음의 값이 된다.

☞ 부동소수점의 저장방식에서의 오차

① 순환소수에 의한 오차

② 유효정밀도에 의한 오차

FLP-02. 정확한 계산이 필요할 때에는 부동소수점 수를 배제할 수 있는지를 고려하라.

컴퓨터에서의 숫자의 표현은 그 크기가 제한되어 있다. 그러므로 1/3과 같은 반복되는 이진 표기값을 정확하게 표현하는 것은 대부분의 부동소수점 표현으로는 불가능하다.

[문제코드]

```
#include <stdio.h>

float mean(float *, size_t);

int main()
{
    int i;
    float array[10];
    float total;

    for (i=0 ; i<sizeof(array)/sizeof(array[0]) ; i++)
    {
        array[i] = 10.1F;
    }

    total = mean(array, sizeof(array)/sizeof(array[0]));
    printf("total = %.10f\n", total);

    getchar();
    return 0;
}

float mean(float *ary, size_t len)
{
    float total = 0.0f;
    int i;

    for (i=0 ; i<len ; i++)
    {
        total += *(ary + i);
        printf("ary[%d] : %f\t total = %f\n", i, *(ary + i), total);
    }

    if (len != 0)
    {
        return total / len;
    }
    else
    {
        return 0.0F;
    }
}
```

[해결방법] 부동소수점수를 정수로 바꿔서 해결한다.

```
#include <stdio.h>
#include <limits.h>

float mean(int *, size_t);

int main()
{
    int i;
    int array[10];
    float total;

    for (i=0 ; i<sizeof(array)/sizeof(array[0]) ; i++)
    {
        array[i] = 101;
    }

    total = mean(array, sizeof(array)/sizeof(array[0]));
    printf("total = %.10f\n", total);

    getchar();
    return 0;
}

float mean(int *ary, size_t len)
{
    int total = 0;
    int i;

    for (i=0 ; i<len ; i++)
    {
        total += *(ary + i);
        printf("ary[%d] : %d\t total = %d\n", i, *(ary + i), total);
    }

    if (len != 0)
    {
        return (float)total / len;
    }
    else
    {
        return 0;
    }
}
```

FLP-30. 부동소수점 변수를 루프 카운터로 사용하지 마라.

- 부동소수점 수는 간단한 십진분수도 정확하게 표현하지 못할 수 도 있다.
- 큰 부동소수점 값에 증가 연산을 적용하면, 경우에 따라서 가능한 정밀도의 한계로 인해 값이 전혀 변하지 않을 수 도 있다.

[문제코드] 정확하게 10회 반복이 안된다. 해결방법은?

```
#include <stdio.h>

int main()
{
    float x;
    int count = 1;
    for (x=0.1f ; x<=1.0f ; x+=0.1f)
    {
        printf("%d. x = %.10f\n", count++, x);
    }

    getchar();
    return 0;
}
```

[문제코드] 자신의 정밀도로 표현이 안되는 작은 값으로 증가시켜서 값이 아예 변하지 않아 무한 반복에 빠질수도 있다.

```
#include <stdio.h>

int main()
{
    float x;
    int count = 1;

    for (x=100000001.0f ; x<=1000000010.0f ; x+=1.0f)
    {
        printf("%d. x = %.30f\n", count++, x);
    }

    getchar();
    return 0;
}
```

FLP-33. 부동소수점 연산용 정수는 먼저 부동소수점으로 바꿔라.

- 계산 시 정수를 사용해 부동소수점 변수에 값을 할당하는 경우, 정보가 손실될 수 있다.

[문제코드] 부동소수점 변수에 저장하기 전에 정수 연산 시에 손실이 발생한다.

```
#include <stdio.h>
#include <float.h>

int main()
{
    short a = 533;
    int b = 6789;
    long c = 466438237;

    float d = a / 7;           // d는 76.0
    double e = b / 30;         // e는 226.0
    double f = c * 255;        // f는 오버플로우되어 음수일 수 있다.

    printf("d : %f\n", d);
    printf("d : %f\n", e);
    printf("d : %f\n", f);

    getchar();
    return 0;
}
```

[해결방법] 정수값을 부동소수점 변수에 저장하여 부동소수점으로 변환한 후, 연산수행

```
#include <stdio.h>
#include <float.h>

int main()
{
    short a = 533;
    int b = 6789;
    long c = 466438237;

    float d = a;
    double e = b;
    double f = c;

    d /= 7;
    e /= 30;
    f *= 255;

    printf("d : %f\n", f);
    printf("e : %f\n", d);
}
```

```
    printf("f : %f\n", e);  
    getchar();  
    return 0;  
}
```

FLP-34. 변환된 값을 저장할 수 있는 자료형인가를 확인하라.

부동소수점 값이 더 작은 범위나 정밀도를 가진 부동소수점 값으로 변환되거나 정수로 변환되는 경우, 혹은 정수가 부동소수점 수로 변환되는 경우, 데이터는 변환될 타입으로 표현가능한 값이어야 한다.

[문제코드] 부동소수점 수를 정수형 변수에 저장할 때, 정수부에 대한 저장을 보장받을 수 없다.

```
#include <stdio.h>

int main()
{
    float f1;
    int i1;

    f1 = 3.14E+10;
    i1 = f1;

    printf("f1 : %f\n", f1);
    printf("i1 : %d\n", i1);

    getchar();
    return 0;
}
```

[해결방법] 부동소수점의 값이 int형의 범위 안의 값인지를 검사한다.

```
#include <stdio.h>
#include <limits.h>

int main()
{
    float f1;
    int i1;

    f1 = 3.14E+10;
    if (f1 > (float)INT_MAX || f1 < (float)(INT_MIN))
    {
        printf("int형에 저장할 수 있는 범위 밖의 수입니다!!!\n");
        /* 에러처리 */
    }
    else
    {
        i1 = f1;
        printf("f1 : %f\n", f1);
        printf("i1 : %d\n", i1);
    }
}
```

```

    }

    getchar();
    return 0;
}

```

[문제코드] 변환하고자 하는 대상이 변환될 타입의 범위 밖의 값일 수 있다.

```

#include <stdio.h>

int main()
{
    float f1;
    double d1;

    d1 = 1.7E+300;
    f1 = (float)d1;
    printf("f1 : %E\n", f1);

    getchar();
    return 0;
}

```

[해결방법] 변환되는 값이 새로운 타입으로 표현될 수 있는지 확인한다.

```

#include <stdio.h>
#include <float.h>

int main()
{
    float f1;
    double d1;

    d1 = 1.7E+300;

    if (d1 > (double)FLT_MAX || d1 < (double)FLT_MIN)
    {
        printf("범위 밖의 데이터입니다!!!\n");
        /* 변환오류처리 */
    }
    else
    {
        f1 = (float)d1;
        printf("f1 : %E\n", f1);
    }

    getchar();
}

```



```
    return 0;  
}
```

ARR-01. 배열의 크기를 얻을 때 포인터를 sizeof연산자의 피연산자로 사용하지 마라.

- sizeof 연산자는 피연산자의 크기를 바이트 단위로 계산해준다.
- sizeof 연산자로 배열의 크기를 결정하기 위해서 사용할 때에는 주의가 필요하다.

[문제코드] 매개변수로 전달된 배열(포인터)에 대해 sizeof 연산을 수행하고 있다.

```
#include <stdio.h>

void clear(int []);

int main()
{
    size_t i;
    int array[5];

    clear(array);
    for (i=0 ; i<sizeof(array)/sizeof(array[0]) ; i++)
    {
        printf("array[%u] : %d\n", i, array[i]);
    }

    getchar();
    return 0;
}

void clear(int array[])
{
    size_t i;
    for (i=0 ; i<sizeof(array)/sizeof(array[0]) ; i++)
    {
        array[i] = 0;
    }
}
```

[해결방법] 배열이 선언된 블록에서 크기를 계산한 후, 인자로 함께 넘긴다.

```
#include <stdio.h>

void clear(int [], size_t);

int main()
{
    size_t i;
    int array[5];
    size_t len = sizeof(array)/sizeof(array[0]);

    clear(array, len);
    for (i=0 ; i<len ; i++)
    {
        printf("array[%u] : %d\n", i, array[i]);
    }
}
```

```
    }

    getchar();
    return 0;
}

void clear(int array[], size_t len)
{
    size_t i;
    for (i=0 ; i<len ; i++)
    {
        array[i] = 0;
    }
}
```

ARR-30. 배열의 인덱스가 유효한 범위 안에 있음을 보장하라.

배열의 참조가 배열의 경계 안에서 일어나게 하는 일은 전적으로 프로그래머의 책임이다.

[문제코드] 아래의 예제에서는 배열 위쪽의 경계만을 보장한다.

```
#include <stdio.h>
#include <stdlib.h>

enum { TABLESIZE=10 };
int *table = NULL;
int insert_in_table(int, int);

int main()
{
    if (insert_in_table(-5, 100) != 0)
    {
        printf("배열에 데이터를 저장하였습니다.\n");
    }
    else
    {
        printf("데이터 저장에 실패하였습니다.");
    }

    getchar();
    return 0;
}

int insert_in_table(int pos, int value)
{
    if (!table)
    {
        table = (int *)malloc(sizeof(int) * TABLESIZE);
        if (table == NULL)
        {
            printf("동적메모리할당실패!!!\n");
            exit(1);
        }
    }

    if (pos >= TABLESIZE)
    {
        return 0;
    }

    table[pos] = value;
    return -1;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdlib.h>

enum { TABLESIZE=10 };
int *table = NULL;
int insert_in_table(size_t, int);

int main()
{
    if (insert_in_table(-5, 100) != 0)
    {
        printf("배열에 데이터를 저장하였습니다.\n");
    }
    else
    {
        printf("데이터 저장에 실패하였습니다.");
    }

    getchar();
    return 0;
}

int insert_in_table(size_t pos, int value)
{
    if (!table)
    {
        table = (int *)malloc(sizeof(int) * TABLESIZE);
        if (table == NULL)
        {
            printf("동적메모리할당실패!!!\n");
            exit(1);
        }
    }

    if (pos >= TABLESIZE)
    {
        return 0;
    }

    table[pos] = value;
    return -1;
}
```

ARR-32. 가변 배열에서 크기를 나타내는 인자가 유효한 범위에 있음을 보장하라.

○ 가변배열은 전통적인 C배열과 근본적으로 동일하다. 주요한 차이점은 선언시 그 크기가 상수표기로 주어지지 않는다는 점이다. 가변배열은 다음과 같이 선언될 수 있다.

```
char val[s];
```

정수 *s*와 배열의 선언이 모두 런타임에서 평가된다.

가변배열에 주어지는 크기 값이 정상적이지 않을 경우, 프로그램은 기대하지 않은 방식으로 동작할 수 있다.

[문제코드] 배열의 크기로 사용되어지는 변수 *s*의 값이 유효한 크기인지 확실치가 않다. *s*가 음수라면 프로그램 프로그램 스택이 깨지고, 너무 큰 양수이면 스택 오버플로우가 발생할 수 있다.

```
#include <stdio.h>

void func(size_t s)
{
    int val[s];
    val[s-1] = 123;
    printf("val[%d] = %d\n", s-1, val[s-1]);
}

int main()
{
    func(10);

    getchar();
    return 0;
}
```

[해결방법] *s*가 유효한 범위의 값인지를 확인한다.

```
#include <stdio.h>

enum { MAX_ARRAY = 1024 };

void func(size_t s)
{
    if (s == 0 || s > MAX_ARRAY)
    {
```

```

        printf("범위지정오류!!!\n");
        return;
    }

    int val[s];
    val[s-1] = 123;
    printf("val[%d] = %d\n", s-1, val[s-1]);
}

int main()
{
    func(10);

    getchar();
    return 0;
}

```

ARR-33. 충분한 크기의 공간에서 복사가 진행됨을 보장하라.

모든 데이터를 담을 수 있을 만큼 크지 않은 배열에 데이터를 복사하면 버퍼 오버플로우를 발생시킬 수 있다.

[문제코드] memcpy() 함수를 이용하여 src가 가리키는 문자열을 dest에 복사함에 있어서 복사하고자 하는 문자열의 수를 src의 크기로 복사하는 실수를 범하고 있다.

```
#include <stdio.h>

enum { WORKSPACE_SIZE=10 };

void func(const char src[], size_t len)
{
    char dest[WORKSPACE_SIZE];
    memcpy(dest, src, len * sizeof(char));
    /* 복사된 dest를 사용하는 코드가 여기에 */
    printf("복사된 문자열 : %s\n", src);
    printf("복사한 문자열 : %s\n", dest);
}

int main()
{
    char string[] = "do not edit this string...";
    func(string, sizeof(string)/sizeof(string[0]));

    getchar();
    return 0;
}
```

[해결방법] 복사가 가능한 크기인가를 확인한다.

```
#include <stdio.h>

enum { WORKSPACE_SIZE=10 };

void func(const char src[], size_t len)
{
    char dest[WORKSPACE_SIZE];

    if (len > WORKSPACE_SIZE)
    {
        printf("문자열의 길이가 너무 길어 복사가 불가능합니다.\n");
        return;
    }
    memcpy(dest, src, len * sizeof(char));
    /* 복사된 dest를 사용하는 코드가 여기에 */
    printf("복사된 문자열 : %s\n", src);
    printf("복사한 문자열 : %s\n", dest);
}
```

```
}  
  
int main()  
{  
    char string[] = "do not edit this string..";  
    func(string, sizeof(string)/sizeof(string[0]));  
  
    getchar();  
    return 0;  
}
```

ARR-34. 표현식에서 배열 타입이 호환 가능함을 보장하라.

표현식에서 호환되지 않는 두 가지 이상의 배열을 사용하려면 정의되지 않은 동작을 초래한다.

[문제코드] 함수 호출 시 넘기는 인자와 인자를 받아 저장하는 파라미터의 타입이 다르다.

```
#include <stdio.h>
#include <stdio_ext.h>

int getMenu(char (*mStr)[10], size_t mCnt)
{
    int i;
    int menuNum = 0;

    while (menuNum != mCnt)
    {
        printf("작업메뉴를 선택하세요.\n");
        for (i=0 ; i<mCnt ; i++)
        {
            printf("%d. %s\n", i+1, *(mStr + i));
        }

        __fpurge(stdin);
        printf("메뉴번호입력 : ");
        if (scanf("%d", &menuNum) != 1) continue;
        if (menuNum < 0 || menuNum > mCnt) continue;
        break;
    }
    return menuNum;
}

int main()
{
    int menuNum = 0;
    char *menuStr[] = { "INPUT", "OUTPUT", "EXIT" };
    size_t menuCnt = sizeof(menuStr)/sizeof(menuStr[0]);

    menuNum = getMenu(menuStr, menuCnt);
    printf("%d번을 선택하셨습니다.\n", menuNum);

    getchar();
    return 0;
}
```

[해결방법] 동일한 자료형으로 일치시킨다.

```
#include <stdio.h>
#include <stdio_ext.h>

int getMenu(char *mStr[], size_t mCnt)
{
    int i;
    int menuNum = 0;

    while (menuNum != mCnt)
    {
        printf("작업메뉴를 선택하세요.\n");
        for (i=0 ; i<mCnt ; i++)
        {
            printf("%d. %s\n", i+1, *(mStr + i));
        }

        __fpurge(stdin);
        printf("메뉴번호입력 : ");
        if (scanf("%d", &menuNum) != 1) continue;
        if (menuNum < 0 || menuNum > mCnt) continue;
        break;
    }
    return menuNum;
}

int main()
{
    int menuNum = 0;
    char *menuStr[] = { "INPUT", "OUTPUT", "EXIT" };
    size_t menuCnt = sizeof(menuStr)/sizeof(menuStr[0]);

    menuNum = getMenu(menuStr, menuCnt);
    printf("%d번을 선택하셨습니다.\n", menuNum);

    getchar();
    return 0;
}
```

ARR-36. 같은 배열을 참조하고 있지 않다면 두 개의 포인터를 빼거나 비교하지 마라.

- 두개의 포인터로 뺄셈을 수행하려면 두 포인터가 같은 배열을 참조하거나 적어도 배열의 마지막 원소 다음 부분을 가리켜야 한다.

- 두 개의 포인터로 뺄셈 연산을 수행하면, 두 원소간의 거리가 계산되는 데, 이때 두원소간의 거리란 두 원소간의 첨자(index) 차이이다.

```
int nums[SIZE];
int *next_num_ptr = nums;
int free_bytes;

/* 배열을 채우면서 next_num_ptr을 증가시킨다. */
free_bytes = (next_num_ptr - nums) * sizeof(int);
printf("free_bytes : %d\n", free_bytes);
```

STR-01. 문자열 관리를 위해 일관된 계획을 사용하여 일관되게 구현하라.

○ 프로그램에서 NULL문자로 종료되는 바이트 문자열을 관리하는데 있어서 두 가지 방법이 있는데, 한 프로젝트 내에서는 둘 중에 한가지 방식을 선택하여 문자열을 일관되게 관리하도록 하라.

1. 문자열을 정적으로 할당된 배열을 통해 관리하는 기법
초과되는 데이터는 버려지기 때문에 결과값으로 나오는 문자열은 충분히 검증되어야 한다.
2. 요구되는 만큼 메모리를 동적으로 할당하여 사용하는 기법
입력을 제한하지 않으면 메모리 고갈로 인한 서비스 거부 공격에 사용될 수 있다.

STR-03. NULL문자로 종료된 문자열이 부적절하게 잘리지 않게 하라.

버퍼 오버플로우 취약성을 완화하기 위해 복사되는 바이트의 수를 제한하는 대체 함수들이 제안되곤 하는데 이 함수들은 지정된 제한을 넘는 문자열을 잘라버린다. 이러한 의도하지 않은 잘림은 데이터 손실과 더불어 프로그램의 취약성을 타나내는 원인이 된다.

`strncpy()`, `strncat()`, `fgets()`, `snprintf()`

[해결방법]

```
char *string_data;
char buff[10];

/* string_data 초기화 */
if (string_data == NULL)
{
    printf("No String\n");
}
else if (strlen(string_data) >= sizeof(buff))
{
    printf("too long\n");
}
else
{
    strncpy(buff, string_data, sizeof(buff));
}
```

STR-04. 기본 문자 집합에서는 문자들을 위해 char형을 사용하라.

○ char형은 signed char 혹은 unsigned char와 동일한 범위, 표기를 갖도록 정의한다. 하지만 둘 중 하나와 같다고는 해도 char형은 이 둘과는 분리된 타입이며, 서로 호환 가능하지도 않다.

○ 표준 문자열 처리 함수들과의 호환성을 위해 문자데이터에는 signed나 unsigned가 아닌 일반 char형을 사용하는 방법이 최선이다.

```
#include <stdio.h>
#include <string.h>

int main()
{
    int len;

    char cstr[] = "char string";
    signed char scstr[] = "signed char string";
    unsigned char ucstr[] = "unsigned char string";

    len = strlen(cstr);
    len = strlen(scstr);    // 경고발생
    len = strlen(ucstr);    // 경고발생

    return 0;
}
```

STR-05. 문자열 상수를 가리키는 포인터는 const로 선언하라.

STR-30. 문자열 리터럴을 수정하려고 하지마라.

문자열 리터럴은 상수이므로 const 지정자에 의해 보호되어야 한다.

[문제코드] 아래의 코드는 런타임 에러를 발생시킨다.

```
#include <stdio.h>

int main()
{
    char *c = "Hello";
    c[0] = 'C';
    printf("%s\n", c);

    getchar();
    return 0;
}
```

[해결방법] 컴파일 에러 발생

```
#include <stdio.h>

int main()
{
    const char *c = "Hello";
    c[0] = 'C';
    printf("%s\n", c);

    getchar();
    return 0;
}
```

STR-06. strtok()에서 파싱되는 문자열이 보존된다고 가정하지 마라.

◦ strtok()함수는 처음 호출되면 문자열내의 구분자가 처음 나타나는 부분까지 파싱하고, 구분자를 NULL문자로 바꾼 후, 토큰의 처음 주소를 반환한다. 이후, 다시 strtok()함수를 호출하면 가장 최근에 NULL문자로 바뀐 부분부터 파싱이 시작된다.

◦ strtok()함수는 인자를 수정하므로 원본 문자열은 안전하지가 않다. 원본 문자열을 보존하고 싶다면 문자열의 복사본을 만들어 사용하도록 하여야 한다.

[문제코드]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *token, *path;

    path = getenv("PATH");

    printf("파싱 전\n");
    puts(path);
    puts("\n");

    token = strtok(path, ";");
    puts(token);
    while (token = strtok(0, ";"))
    {
        puts(token);
    }
    puts("\n");

    printf("파싱 후\n");
    puts(path);
    puts("\n");

    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *token, *path, *copy;

    path = getenv("PATH");
    copy = (char *)calloc(strlen(path)+1, sizeof(char));
    if (copy == NULL)
    {
        return 1;
    }
    strncpy(copy, path, strlen(path));

    printf("파싱 전\n");
    puts(path);
    puts("\n");

    token = strtok(copy, ";");
    puts(token);
    while (token = strtok(0, ";"))
    {
        puts(token);
    }
    puts("\n");

    printf("파싱 후\n");
    puts(path);
    puts("\n");

    free(copy);

    getchar();
    return 0;
}
```

STR-35. 경계가 불분명한 소스로부터 고정된 길이의 배열에 데이터를 복사하지 마라.

별도의 경계 없이 복사를 수행하는 함수들은 종종 외부 입력에서 적절한 크기가 들어올 것이라고 생각하여 버퍼 오버플로우를 발생시킬 수 있다.

[문제코드] gets()함수는 입력되는 문자열의 길이를 제한할 수 없으므로 버퍼 오버플로우를 발생시킬 수 있다.

```
#include <stdio.h>
#include <stdio_ext.h>

int main()
{
    char name[10];

    gets(name);
    printf("name : %s\n", name);

    __fpurge(stdin);
    getchar();
    return 0;
}
```

[해결방법] 길이제한이 가능한 fgets()함수를 사용한다.

```
#include <stdio.h>
#include <stdio_ext.h>

int main()
{
    char name[10];

    fgets(name, sizeof(name), stdin);
    printf("name : %s\n", name);

    __fpurge(stdin);
    getchar();
    return 0;
}
```

[문제코드] scanf() 함수를 이용하여 문자열을 입력 시, 길이를 제한할 수 없다.

```
#include <stdio.h>
#include <stdio_ext.h>

int main()
{
    char name[10];

    scanf("%s", name);
    printf("%s\n", name);

    __fpurge(stdin);
    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdio_ext.h>

#define STRING(n) STRING_AGAIN(n)
#define STRING_AGAIN(n) #n
#define READ_CHARS 9

int main()
{
    char name[READ_CHARS + 1];

    scanf("%"STRING(READ_CHARS)"s", name);
    printf("%s\n", name);

    __fpurge(stdin);
    getchar();
    return 0;
}
```

▶ 부실한 메모리 관리로 발생 가능한 문제점

- 힙 버퍼 오버플로우
- 댕글링 포인터 (Dangling pointer)
- 중복해제

※ Dangling pointer : 존재하지 않는 대상을 참조하는 포인터

- 초기화 되지 않은 포인터
- free()함수에 의해 해제된 메모리 공간을 가리키는 포인터

MEM-00. 동일한 함수 또는 동일한 파일 내에서 메모리를 할당하고 해제하라.

[문제코드] 각기 다른 함수나 파일에서 메모리를 할당하고 해제하는 것은 중복해제의 취약성이 발생할 수 있다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum { MIN_SIZE_ALLOWED = 20 };

int verify_list(char *, size_t);
void process_list(size_t);

int main()
{
    process_list(10);

    getchar();
    return 0;
}

int verify_list(char *list, size_t size)
{
    if (size < MIN_SIZE_ALLOWED)
    {
        free(list);
        return -1;
    }
    return 0;
}

void process_list(size_t number)
{
    char *list = (char *)malloc(number);
    if (list == NULL)
    {
        return;
    }
}
    
```

```

    }

    if (verify_list(list, number) == -1)
    {
        printf("할당된 메모리의 크기가 %d 미만 입니다.\n",
MIN_SIZE_ALLOWED);
        free(list);
        return;
    }
    /* list 사용 */
    strcpy(list, "memory test");
    printf("%s\n", list);
    free(list);
}

```

[해결방법]

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum { MIN_SIZE_ALLOWED = 20 };

int verify_list(char *, size_t);
void process_list(size_t);

int main()
{
    process_list(10);

    getchar();
    return 0;
}

int verify_list(char *list, size_t size)
{
    if (size < MIN_SIZE_ALLOWED)
    {
        //free(list);
        return -1;
    }
    return 0;
}

void process_list(size_t number)
{
    char *list = (char *)malloc(number);
    if (list == NULL)
    {
        return;
    }

    if (verify_list(list, number) == -1)
    {

```

```

        printf("할당된 메모리의 크기가 %d 미만 입니다.\n",
MIN_SIZE_ALLOWED);
        free(list);
        return;
    }
    /* list 사용 */
    strcpy(list, "memory test");
    printf("%s\n", list);
    free(list);
}

```

MEM-01. free()후 즉시 포인터에 새로운 값을 저장하라.

댕글링 포인터는 중복해제나 해제된 메모리에 액세스 하는 취약성이 있다.

[문제코드]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h>

enum { STR_SIZE_MAX=20 };
enum { TYPE_1=0, TYPE_2=0 };
char *getMessage();

int main()
{
    char *message;
    size_t message_type;

    message = getMessage();
    if (message == NULL)
    {
        return 1;
    }

    printf("message_type (0 or 1) : ");

    if (scanf("%lu", &message_type) != 1 || message_type > 2)
    {
        printf("입력오류\n");
        return 2;
    }

    if (message_type == TYPE_1)
    {
        /* message를 사용하는 코드 */
        free(message);
    }

    if (message_type == TYPE_2)
    {
        /* message를 사용하는 코드 */
        free(message);
    }

    __fpurge(stdin);
    getchar();
    return 0;
}

char *getMessage()
{

```



```

char tmp[STR_SIZE_MAX] = "";
char *nr, *message;

memset(tmp, '\0', sizeof(tmp));
printf("메세지 입력 (20바이트 이내) : ");

if (fgets(tmp, sizeof(tmp), stdin) == NULL)
{
    return NULL;
}

nr = strchr(tmp, '\n');
if (nr != NULL) { *nr = '\0'; }
if (tmp[0] == '\0')
{
    return NULL;
}

message = (char *)malloc(strlen(tmp) + 1);
if (message == NULL)
{
    return NULL;
}

strncpy(message, tmp, strlen(tmp));
message[strlen(tmp)] = '\0';
return message;
}

```

[해결방법] 포인터에 할당되었던 메모리를 해제한 후에는 다른 유효한 객체를 참조하게 하거나 NULL값을 지정한다.

※ free()함수는 NULL포인터를 인자로 받는 경우, 아무 동작도 하지 않는다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h>

enum { STR_SIZE_MAX=20 };
enum { TYPE_1=0, TYPE_2=0 };
char *getMessage();

int main()
{
    char *message;
    size_t message_type;

    message = getMessage();
    if (message == NULL)
    {
        return 1;
    }

    printf("message_type (0 or 1) : ");
}

```

```

__fpurge(stdin);
if (scanf("%lu", &message_type) != 1 || message_type > 2)
{
    printf("입력오류\n");
    return 2;
}

if (message_type == TYPE_1)
{
    /* message를 사용하는 코드 */
    free(message);
    message = NULL;
}

if (message_type == TYPE_2)
{
    /* message를 사용하는 코드 */
    free(message);
    message = NULL;
}

__fpurge(stdin);
getchar();
return 0;
}

char *getMessage()
{
    char tmp[STR_SIZE_MAX] = "";
    char *nr, *message;

    memset(tmp, '\0', sizeof(tmp));
    printf("메세지 입력 (20바이트 이내) : ");
    __fpurge(stdin);
    if (fgets(tmp, sizeof(tmp), stdin) == NULL)
    {
        return NULL;
    }


    nr = strchr(tmp, '\n');
    if (nr != NULL) { *nr = '\0'; }
    if (tmp[0] == '\0')
    {
        return NULL;
    }

    message = (char *)malloc(strlen(tmp) + 1);
    if (message == NULL)
    {
        return NULL;
    }

    strncpy(message, tmp, strlen(tmp));
    message[strlen(tmp)] = '\0';
    return message;
}

```

}

 예외) 함수나 block내의 auto변수는 함수가 종료되거나 블록을 탈출하면서 자동으로 소멸되므로 NULL값을 지정할 필요가 없다.

MEM-02. 메모리 할당 함수의 반환값을 즉시 할당된 타입의 포인터로 변환시켜라.

[문제코드] 메모리 할당 함수의 반환값은 void *형이므로 변환없이 사용할 경우, 실제 할당된 메모리와 다른 크기의 메모리를 사용하게 되는 경우가 발생한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h>

typedef struct _gadget
{
    int i;
    double d;
} gadget;

typedef struct _widget
{
    char c[8];
    int i;
    double d;
} widget;

int main()
{
    widget *p;
    p = malloc(sizeof(gadget));
    if (p == NULL)
    {
        return 1;
    }

    p->i = 0;
    p->d = 0.0;
    strcpy(p->c, "apple");
    printf("p->c : %s\n", p->c);
    printf("p->i : %d\n", p->i);
    printf("p->d : %.2lf\n", p->d);
    free(p);
    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio_ext.h>

typedef struct _gadget
```

```
{
    int i;
    double d;
} gadget;

typedef struct _widget
{
    char c[8];
    int i;
    double d;
} widget;

int main()
{
    widget *p;

    /*      잘못된 주소 타입 저장으로 경고 발생
       p = (gadget *)malloc(sizeof(gadget));
       바른 할당과 바른 주소 저장으로 오류 없음
       p = (widget *)malloc(sizeof(widget));
    */
    p = (widget *)malloc(sizeof(widget));
    if (p == NULL) { return 1; }

    p->i = 0;
    p->d = 0.0;
    strcpy(p->c, "apple");
    printf("p->c : %s\n", p->c);
    printf("p->i : %d\n", p->i);
    printf("p->d : %.2lf\n", p->d);

    free(p);
    getchar();
    return 0;
}
```

▶ 자주 사용할 메모리 동적할당 매크로

```
#define MALLOC(number, type) ((type *)malloc((number) * sizeof(type)))

#define CALLOC(number, type) ((type *)calloc((number), sizeof(type)))

#define REALLOC(pointer, number, type) \
    ((type *)realloc(pointer, (number)*sizeof(type)))

#define ALLOC_ERROR_CHECK(pointer) \
    if (pointer == NULL) { printf("메모리할당에러\n"); exit(0); }
```

MEM-03. 재사용을 위해 반환하는 리소스에 있는 중요한 정보를 초기화 하라.

[문제코드] 재사용하게 된 리소스에 민감한 데이터가 들어가 있는 경우, 사용권한이 없는 사용자에게 해당 데이터가 노출될 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MALLOC(number, type) \
    ((type *)malloc((number) * sizeof(type)))
#define ALLOC_ERROR_CHECK(pointer) \
    if (pointer == NULL) \
    { \
        printf("메모리할당오류\n"); \
        exit(1); \
    }

int main()
{
    char *secret;
    char *new_secret;
    size_t size;

    secret = MALLOC(15, char);
    ALLOC_ERROR_CHECK(secret);
    strcpy(secret, "800215-1194919");
    printf("secret의 내용 : %s\n", secret);

    size = strlen(secret);
    new_secret = MALLOC(size+1, char);
    ALLOC_ERROR_CHECK(new_secret);
    strcpy(new_secret, secret);
    printf("new_secret의 내용 : %s\n", new_secret);

    free(new_secret);
}
```

```

    new_secret = NULL;
    free(secret);
    secret = NULL;

    getchar();
    return 0;
}

```

[해결방법]

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MALLOC(number, type) \
    ((type *)malloc((number) * sizeof(type)))
#define ALLOC_ERROR_CHECK(pointer) \
    if (pointer == NULL) \
    { \
        printf("메모리할당오류\n"); \
        exit(1); \
    }

int main()
{
    char *secret;
    char *new_secret;
    size_t size;

    secret = MALLOC(15, char);
    ALLOC_ERROR_CHECK(secret);
    strcpy(secret, "800215-1194919");
    printf("secret의 내용 : %s\n", secret);

    size = strlen(secret);
    new_secret = MALLOC(size+1, char);
    ALLOC_ERROR_CHECK(new_secret);
    strcpy(new_secret, secret);
    printf("new_secret의 내용 : %s\n", new_secret);

    memset( (volatile char *)new_secret, '\\0', size);
    free(new_secret);
    new_secret = NULL;
    memset( (volatile char *)secret, '\\0', size);
    free(secret);
    secret = NULL;

    getchar();
    return 0;
}

```

realloc()함수를 이용하여 할당된 메모리 공간을 확장 시, 만약 현재 할당된 공간뒤에 확장 되는 공간만큼의 여분의 공간이 없는경우, realloc()함수는 새로운 위치에 공간을 할당하여 기존의 데이터를 복사하고, 기존의 메모리공간을 해제한 후, 새로 할당된 메모리의 시작주소를 반환한다.

이때 기존에 사용하던 메모리 공간을 해제하지만 데이터 초기화까지 실행되지는 않으므로 중요한 데이터가 그대로 남은 채, 악용될 수 있다.

[문제코드]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MALLOC(number, type) \
    ((type *)malloc((number) * sizeof(type)))
#define REALLOC(pointer, number, type) \
    ((type *)realloc(pointer, (number) * sizeof(type)))
#define ALLOC_ERROR_CHECK(pointer) \
    if (pointer == NULL) \
    { \
        printf("메모리할당오류\n"); \
        exit(1); \
    }

int main()
{
    char *secret;
    size_t size;

    secret = MALLOC(7, char);
    ALLOC_ERROR_CHECK(secret);
    strcpy(secret, "800215");
    printf("secret의 내용 : %s\n", secret);

    size = strlen(secret);
    REALLOC(secret, size+8+1, char);
    ALLOC_ERROR_CHECK(secret);
    strcat(secret, "-1194919");
    printf("secret의 내용 : %s\n", secret);

    size = strlen(secret);
    memset( (volatile char *)secret, '\0', size);
    free(secret);
    secret = NULL;

    getchar();
    return 0;
}
```

[해결방법] realloc()함수에 의존하지 않고 realloc()함수와 비슷하게 동작하는 로직을 만들어서 해결하라.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MALLOC(number, type) \
    ((type *)malloc((number) * sizeof(type)))
#define REALLOC(pointer, number, type) \
    ((type *)realloc(pointer, (number) * sizeof(type)))
#define ALLOC_ERROR_CHECK(pointer) \
    if (pointer == NULL) \
    { \
        printf("메모리할당오류\n"); \
        exit(1); \
    }

int main()
{
    char *secret;
    char *tmp_buff;
    size_t size;

    secret = MALLOC(7, char);
    ALLOC_ERROR_CHECK(secret);
    strcpy(secret, "800215");
    printf("secret의 내용 : %s\n", secret);

    size = strlen(secret);
    tmp_buff = MALLOC(size+8+1, char);
    ALLOC_ERROR_CHECK(tmp_buff);
    memcpy(tmp_buff, secret, size+1);
    strcat(tmp_buff, "-1194919");

    memset( (volatile char *)secret, '\0', size);
    free(secret);
    secret = tmp_buff;
    tmp_buff = NULL;
    printf("secret의 내용 : %s\n", secret);

    memset( (volatile char *)secret, '\0', size);
    free(secret);

    getchar();
    return 0;
}

```

MEM-05. 큰 스택 할당을 피하라.

○ 문제점

특히 스택의 증가가 공격에 의해 제어되거나 영향 받을 수 있는 경우라면 더욱 피해야 한다.

○ 해결방법

가변배열을 사용하기 보다 메모리를 동적으로 할당하여 사용한다.

○ 문제점

재귀 함수 역시 스택 할당 초과를 초래할 수 있다. 재귀 함수들은 과도한 재귀 호출 동작으로 인해 스택을 고갈시켜버리지 않는지를 반드시 보장하여야 한다.

○ 해결방법

재귀 함수를 사용하지 않고, 반복문을 이용하여 스택 할당 초과 문제를 해결한다.

MEM-08. 동적으로 할당된 배열을 resize 하는 경우에만 realloc()함수를 사용하라.

realloc()으로 재할당되는 배열은 동일한 타입이어야 하는데 다른 타입의 기억공간으로 재할당하면 realloc()함수예외에 복사된 데이터들이 제대로 사용될 수 없다.

[문제코드]

```
#include <stdio.h>
#include <stdlib.h>

enum { ALLOC_LEN=2 };

int main()
{
    int *iptr;
    short *sptr;

    iptr = (int *)malloc(ALLOC_LEN * sizeof(int));
    if (iptr == NULL)
    {
        return 1;
    }
    *(iptr + 0) = 0xAAAABBBB;
    *(iptr + 1) = 0xCCCCDDDD;

    sptr = (short *)realloc(iptr, ALLOC_LEN * 2 * sizeof(int));
    if (sptr == NULL)
    {
        return 1;
    }

    printf("sptr[0] : %hX\n", *(sptr+0));
    printf("sptr[1] : %hX\n", *(sptr+1));
    printf("sptr[2] : %hX\n", *(sptr+2));
    printf("sptr[3] : %hX\n", *(sptr+3));

    getchar();
    return 0;
}
```

[해결방법]

```
#include <stdio.h>
#include <stdlib.h>

enum { ALLOC_LEN=2 };

int main()
{
    int *iptr;
    int *sptr;

    iptr = (int *)malloc(ALLOC_LEN * sizeof(int));
    if (iptr == NULL)
```

```

{
    return 1;
}
*(iptr + 0) = 0xAAAABBBB;
*(iptr + 1) = 0xCCCCDDDD;

sptr = (int *)realloc(iptr, ALLOC_LEN * 2 * sizeof(int));
if (sptr == NULL)
{
    return 1;
}

printf("sptr[0] : %X\n", *(sptr+0));
printf("sptr[1] : %X\n", *(sptr+1));
printf("sptr[2] : %X\n", *(sptr+2));
printf("sptr[3] : %X\n", *(sptr+3));

getchar();
return 0;
}

```

MEM-09. 메모리 할당 루틴이 메모리를 초기화 해줄 것이라고 가정하지 마라.

malloc()함수로 할당된 메모리나, realloc()함수로 할당된 메모리 중 추가된 메모리의 경우, 초기화 되어 있지 않거나 다른 섹션(혹은 다른 프로그램)에서 사용하던 데이터를 가지고 있을 수 도 있다.

[문제코드] malloc()함수로 할당된 메모리에 strncpy()함수를 이용하여 문자열을 복사하면 문자열의 끝에 NULL문자가 자동으로 저장되지 않는다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_BUF_SIZE = 256 };

int main()
{
    char *str = "dream & hope";
    char *buf;
    size_t len;

    len = strlen(str);
    if (len == 0 || len >= MAX_BUF_SIZE - 1)
    {
        printf("문자열의 길이가 0이거나 너무 길어서 처리할 수
없습니다.\n");
        return 1;
    }

    buf = (char *)malloc(MAX_BUF_SIZE * sizeof(char));
    if (buf == NULL)
    {
        return 1;
    }

    /* strncpy() 함수는 지정된 바이트수 만큼 복사한 후,
    NULL을 추가하지 않는다. */
    strncpy(buf, str, len);
    printf("buf : %s\n", buf);
    free(buf);
    getchar();
    return 0;
}
```

[해결방법 1] 문자열을 복사한 후, 명시적으로 NULL문자를 추가한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_BUF_SIZE = 256 };

int main()
{
    char *str = "dream & hope";
    char *buf;
    size_t len;

    len = strlen(str);
    if (len == 0 || len >= MAX_BUF_SIZE - 1)
    {
        printf("문자열의 길이가 0이거나 너무 길어서 처리할 수
없습니다.\n");
        return 1;
    }

    buf = (char *)malloc(MAX_BUF_SIZE * sizeof(char));
    if (buf == NULL)
    {
        return 1;
    }

    strncpy(buf, str, len);
    /* 문자열을 복사한 후, NULL을 추가 */
    *(buf + len) = '\0'; // buf[len] = '\0';
    printf("buf : %s\n", buf);
    free(buf);
    getchar();
    return 0;
}
```

[해결방법 2] malloc()함수에 의해 할당받은 메모리 공간을 사용하기전에 NULL로 초기화 한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_BUF_SIZE = 256 };

int main()
{
    char *str = "dream & hope";
    char *buf;
    size_t len;

    len = strlen(str);
    if (len == 0 || len >= MAX_BUF_SIZE - 1)
    {
        printf("문자열의 길이가 0이거나 너무 길어서 처리할 수
없습니다.\n");
        return 1;
    }

    buf = (char *)malloc(MAX_BUF_SIZE * sizeof(char));
    if (buf == NULL)
    {
        return 1;
    }

    /* malloc()함수에 의해 할당된 공간을 NULL로 초기화 */
    memset(buf, '\0', MAX_BUF_SIZE);
    strncpy(buf, str, len);
    printf("buf : %s\n", buf);
    free(buf);
    getchar();
    return 0;
}
```

MEM-30. 해제된 메모리에 접근하지 마라.

이미 해제된 메모리에 접근하면 heap상의 데이터 구조가 손상될 수 있다.

[문제코드]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct _node Node;
typedef struct _list List;

struct _node
{
    Node *next;
    char name[20];
    int age;
};

struct _list
{
    Node *head;
};

int add_node(List *, char *, int);
void print_node(List *);
void destroy(List *);

int main()
{
    List *list;
    list = (List *)calloc(1, sizeof(List));
    if (list == NULL)
    {
        return 1;
    }
    list->head = (Node *)calloc(1, sizeof(Node));
    if (list->head == NULL)
    {
        return 1;
    }

    add_node(list, "김기희", 20);
    add_node(list, "홍길동", 19);
    add_node(list, "심청이", 16);
    print_node(list);

    destroy(list);
    getchar();
    return 0;
}
```



```

int add_node(List *list, char *name, int age)
{
    Node *tmp;
    Node *node;
    node = (Node *)calloc(1, sizeof(Node));
    if (node == NULL)
    {
        printf("노드생성실패!\n");
        return 0;
    }
    strcpy(node->name, name);
    node->age = age;

    for (tmp = list->head ; tmp->next != NULL ; tmp = tmp->next);
    tmp->next = node;

    return 1;
}

void print_node(List *list)
{
    Node *tmp;

    for (tmp = list->head->next ; tmp != NULL ; tmp = tmp->next)
    {
        printf("이름 : %s\n", tmp->name);
        printf("연령 : %d\n", tmp->age);
    }
}

void destroy(List *list)
{
    Node *p;
    for (p = list->head->next ; p != NULL ; p = p->next)
    {
        free(p);
    }
}

```

[해결방법]

```

void destroy(List *list)
{
    Node *p, *tmp;

    for (p = list->head->next ; p != NULL ; p = tmp)
    {
        tmp = p->next;
        free(p);
    }
}

```

MEM-31. 동적으로 할당된 메모리는 한 번만 해제하라.

- 메모리를 여러번 해제하면 해제된 메모리에 접근하는 것과 같은 결과를 초래한다.
(double free)
- 힙 상의 데이터 구조가 손상되고 프로그램의 보안에 취약성을 유발한다.
- double free의 취약성을 제거하려면 동적 메모리가 정확히 한 번만 해제되도록 보장해야 한다.